Security Protocols: Specification, Verification, Implementation, and Composition

Almousa, Omar

Publication date:
2016

Document Version
Publisher's PDF, also known as Version of record

Security Protocols: Specification, Verification, Implementation, and Composition

Omar Saad Kulieb Almousa

Kongens Lyngby 2015
PhD-2015-391

Technical University of Denmark
Department of Applied Mathematics and Computer Science
Richard Petersens Plads, building 324,
2800 Kongens Lyngby, Denmark
Phone +45 4525 3031
compute@compute.dtu.dk
www.compute.dtu.dk PhD-2015-391

SSN:0909-3192

To all Rogers!

Summary (English)

An important aspect of Internet security is the security of cryptographic protocols that it deploys. We need to make sure that such protocols achieve their goals, whether in isolation or in composition, i.e., security protocols must not suffer from any flaw that enables hostile intruders to break their security. Among others, tools like OFMC [MV09b] and Proverif [Bla01] are quite efficient for the automatic formal verification of a large class of protocols. These tools use different approaches such as symbolic model checking or static analysis. Either approach has its own pros and cons, and therefore, we like to combine their strengths. Moreover, we need to ensure that the protocol implementation coincides with the formal model that we verify using such tools.

This thesis shows that we can simplify the formal verification of protocols in several ways. First, we introduce an Alice and Bob style language called SPS (Security Protocol Specification) language, that enables users, without requiring deep expertise in formal models, to specify a wide range of real-world protocols in a simple and intuitive way. Thus, SPS allows users to verify their protocols using different tools, and generate robust implementations in different languages. Moreover, SPS has the "ultimate" formal semantics for Alice and Bob notation in the presence of an arbitrary set of cryptographic operators and their algebraic theory. Despite its generality, this semantics is mathematically simpler than any previous attempt.

Second, we introduce two types of relative soundness results that reduce complex verification problems into simpler ones. The first kind is typing results show that if a security protocol that fulfills a number of sufficient conditions has an attack then it has a well-typed attack. The second kind considers the parallel composition of protocols, showing that if the parallel composition of two protocols, that fulfill a number of sufficient conditions, allows for an attack then one of the protocols, at least, has an attack in isolation. In fact, we unify and generalize over prior relative soundness results. The most important generalization is the support for all security properties of the geometric fragment proposed by [Gut14].

Summary (Danish)

Et vigtigt aspekt af internetsikkerhed er sikkerheden i de kryptografiske protokoller, der benyttes. Vi er nødt til at sørge for, at sådanne protokoller fungerer korrekt, både hver for sig og i sammensætning. Dvs. sikkerhdesprotokoller må ikke lide af nogen fejl der gør det muligt for hackere til at bryde deres sikkerhed. Blandt andet, værktøjer som OFMC [MV09b] og Proverif [Bla01] er ganske effektive til automatiseret formel verifikation af en stor klasse af protokoller. Disse værktøjer bruger forskellige tilgange, såsom symbolsk model checking eller statisk analyse. Begge tilgange har fordele og ulemper, og derfor er vi nødt til at kombinere deres styrker. Desuden er vi nødt til at sikre, at der er sammenfald mellem protokolimplementeringerne, og de formelle modeller vi kontrollerer ved at bruge disse værktøjer.

Denne afhandling viser, at vi kan forenkle formel verifikation af protokoller på flere måder. Først introducerer vi et sprog, i Alice og Bob stilen, kaldet SPS (Security Protocol Specification). Sproget giver brugere, uden stor ekspertise i formelle modeller, mulighed for at specificere en bred vifte af protokoller fra den virkelige verden på en enkel og intuitiv måde. Således giver SPS brugerne mulighed for at kontrollere deres protokoller ved hjælp af forskellige værktøjer og skabe robuste implementeringer i forskellige sprog. Desuden SPS har den "ultimative" formelle semantik for Alice og Bob notationen i nærværelse af et vilkårligt sæt af kryptografiske operatører og deres algebraiske teori. På trods af sin generalitet er denne semantik matematisk enklere end nogen tidligere forsøg.

Dernæst introducerer vi to slags relativ korrekthed, der kan forsimple komplekse verifikationsproblemer. Den første slags er at skrive resultater, der viser, at hvis en sikkerhedsprotokol, der opfylder en række betingelser, har et angreb så har et vel-typet angreb. Den anden slags betragter den parallelle sammensætning af protokoller, og viser, at hvis den parallelle sammensætning af to protokoller, der opfylder en række tilstrækkelige betingelser, giver mulighed for et angreb, har mindst en ag protololleme, et angreb i isolation. Mere konkret, forener og

generaliserer vi forudgående resultater inde for relativ korrekthed. Den vigtigste generalisering er støtten til alle sikkerhedsegenskaber af geometriske fragment foreslåede af [Gut14].

Preface

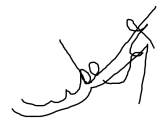
This thesis was prepared at the Department of Applied Mathematics and Computer Science (DTU Compute) of the Technical University of Denmark (DTU), in partial fulfillment of the requirements for acquiring the Ph.D. degree in Computer Science.

The Ph.D. study has been carried out under the supervision of Associate Professor Sebastian Mödersheim and Professor Hanne Riis Nielson in the period from December 2012 to December 2015.

The Ph.D. project was partially funded by the EU FP7 Project no. 318424, "FutureID: Shaping the Future of Electronic Identity" [Fut].

Most of the work behind this dissertation has been carried out independently and I take full responsibility for its contents. A substantial part of the scientific work reported in this thesis is based on joint work with my supervisor, partners from the FutureID project, and Professor Luca Viganò (my supervisor during my external research stay at King's College-London) [Fut13c, AMV15a, AMV15b, AMMV15a, AMMV15b, Fut15].

Lyngby, 14-December-2015



Omar Saad Kulieb Almousa

Acknowledgements

I would like to thank my supervisor Sebastian Mödersheim. I am deeply grateful for his tremendous help, continuous support and kind care. It has been an honour to be his first Ph.D. student.

Special thanks are also due to my co-supervisor Hanne Riis Nielson and my external-stay supervisor Luca Viganò for their practical guidance and continuous support. Many thanks are due to the reviewing committe: Alberto Lluch Lafuente, Achim Brucker, and Christoph Sprenger.

I extend my thanks to all LBT professors, especially Flemming Nielson, Christian Probst; the feedback they gave me during LBT talks improved not only my presentation skills, but also the results of my research.

Many thanks to all of my colleagues in LBT: Alessandro, Hugo, Jans, Lars, Laust, Marieta, Roberto, Ximeng, Zara and Andreas. They have contributed immensely to my personal and professional time at DTU-Compute. I also thank my friends Mohammad, Othman, Qasim, Hasan, and Majdi. Many thanks to Lotte and Cathrin for their help.

I gratefully acknowledge the funding for my Ph.D. the EU FP7 Project no. 318424, "FutureID: Shaping the Future of Electronic Identity".

I would like to thank my parents, my brothers and sisters, my wife and my kids for all their love, patience and encouragement. Thank you.

Lastly, "And the last of their call will be: Praise to Allah, Lord of the worlds!".

Contents

List of Figures

NOTATIONS

f	Function, general	
ch	Channel(cf. Section 2.3)	19
insec	Insecure channel(cf. Section 2.3)	19
S	Operational strand (cf. Section 2.5)	21
fv	Free variable of(cf. Section 2.5)	21
$strand$	Operational strand(cf. Section 2.5)	21
$steps$	Sequence of steps of an operational strand(cf. Section 2.5)	21
$rest$	Rest of steps of an operational strand(cf. Section 2.5)	21
$step$	Step in an operational strand(cf. Section 2.5)	21
request	Event in operational strands(cf. Section 2.5)	21
receive	Receive step in an operational strand(cf. Section 2.5)	21
fresh	Fresh value step in an operational strand(cf. Section 2.5)	21
event	Event in an operational strand(cf. Section 2.5)	21
$X := t$	Strand-macro step in an operational strand(cf. Section 2.5)	21
Σ	Set of a operators, function symbols(cf. Definition 3.1)	28
Σ_0	Set of a protocol constants(cf. Definition 3.1)	28
Σ_p	Set of public operators(cf. Definition 3.1)	28
Σ_m	Set of private operators, mappings(cf. Definition 3.1)	28
Σ_d	Set of destructors(cf. Definition 3.1)	28
Σ_c	Set of constructors(cf. Definition 3.1)	28
$\mathcal{X}$	Label variable(cf. Definition 3.1)	28
$\mathcal{V}$	Set of variables(cf. Definition 3.1)	28
$\mathcal{T}$	Set of terms(cf. Definition 3.1	28
$\mathcal{I}$	Interpretation(cf. Definition 3.1)	28
t, s, m	Term (cf. Definition 3.1)	28
$\mathcal{X}$	Label variable(cf. Definition 3.1)	28
$\mathcal{L}$	Set of label variables(cf. Definition 3.1)	28
$\approx$	Congruence over ground terms(cf. Definition 3.1)	28

i	The intruder (cf. Section 3.5)	46
own	Owner of a strand(cf. Section 4.1)	53
par	Parameters derived from a knowledge (cf. Section 4.1)	53
head	JavaScript code header (cf. Section 4.1)	53
$[\![\cdot]\!]_{JS}$	Translator to JavaScript(cf. Section 4.1)	53
$[\![\cdot]\!]_{\pi}$	Translator to Applied π(cf. Section 4.2)	58
$\mathsf{byte}(n)$	One-byte constant n(cf. Section 5.3)	79
$\cdot$	Concatenation operator(cf. Section 5.3)	79
`offco`	Offset with a fixed length of bytes(cf. Section 5.3)	79
`tlv`	Tag-length-value encoding(cf. Section 5.3)	79
$\hat{\Sigma}$	Set of operators (cf. Section 7.1)	109
$\mathcal{C}$	Set of constants (cf. Section 7.1)	109
$\mathcal{T}_{\Sigma\cup\mathcal{C}}(\mathcal{V})$	Set of terms built with Σ, $\mathcal{C}$ and $\mathcal{V}$(cf. Section 7.1)	109
atomic	Atomic term(cf. Section 7.1)	109
$\mathcal{C}_{pub}$	Set of constants: long-term public constants(cf. Section 7.1)	109
$\mathcal{C}_{priv}$	Set of constants: long-term secrets(cf. Section 7.1)	109
$\mathcal{C}_{P_i}$	Set of protocol P_i constants(cf. Section 7.1)	109
$\mathcal{M}$	Intruder's Knowledge(cf. Definition 7.1)	111
$\Vdash$	Knowledge deduction in the free-algebra(cf. Definition 7.1)	111
`Ana`	Encoding for analysis rules(cf. Definition 7.1)	111
ϕ	A symbolic constraint(cf. Section 8.1)	113
ϕ_σ	A symbolic constraint, sub-language of ϕ(cf. Section 8.1)	113
ϕ	Symbolic constraint(cf. Section 8.1)	113
$\models$	Model relation (cf. Definition 8.1)	114
var	The variables of(cf. Definition 8.1)	114
$\mathcal{S}$	Set of operational strands (cf. Section 8.2)	115
$\Longrightarrow$	Transition relation over symbolic states(cf. Section 8.2)	115
lts	Event to indicates long-term secret (cf. Section 8.2)	115
Ψ	State formulas in the geometric fragment, a goal (cf. Definition 8.3)	117
ψ	Antecedent of a state formula (cf. Definition 8.3)	117
ψ_0	Consequent of a state formula (cf. Definition 8.3)	117
ik	Intruder knows a term (cf. Definition 8.3)	117
$\models_{\mathfrak{S}}$	Models relation for symbolic states(cf. Section 8.3)	117
$tr_{\mathcal{M},E}$	Translating goals to symbolic constraint(cf. Section 8.4)	118
eq	Constraint corresponding to a substitution(cf. Definition 8.5)	120
mgu	Most general unifier(cf. Definition 8.5)	120
simple	Simple constraint(cf. Definition 8.6)	121
size	Size of symbolic constraint (cf. Theorem 8)	124
Σ_i	Sum of the lengths of i knowledges (cf. Theorem 8)	124
$t : \tau$	Term t has the type τ (cf. Section 9.1)	129
$\mathfrak{T}_a$	Set of atomic types (cf. Section 9.1)	129
Γ	Typing function (cf. Definition 9.1)	129
MP_t	Message pattern of a term(cf. Definition 9.2)	131

MP_S	Message patterns of a strand(cf. Definition 9.2)	131
MP_Ψ	Message patterns of a goal(cf. Definition 9.2)	131
MP	Message pattern of a protocol(cf. Definition 9.2)	131
SMP	Sub-message patterns(cf. Definition 9.2)	131
Φ	Symbolic constraint, attack (cf. Theorem 9)	133
dom	Domain, general	
$range$	Range, general	

Chapter 1

Introduction

An important part of Internet security is the security of cryptographic protocols. We need to make sure that such protocols achieve their goals and that they do not suffer from any flaw that enables hostile intruders to break their security (whether in isolation or in composition). Automatic formal verification tools like [Mea96, Low97a, CJM00, Bla01, CMR01, AC04, Cre06, Tur06, MV09b, EMM09] are quite efficient for a large class of security protocols [NS78, Low95, ACC+08]. Those tools use different approaches, i.e., symbolic model checking or static analysis. Either approach has its own pros and cons, and therefore we need to combine their strengths. Moreover, we like to ensure that the protocol implementations coincide with the formal models that we verify using such tools.

One goal of this work is the definition of an Alice-and-Bob style language that is completely independent of the approach of the verification tools. Alice-and-Bob notation is a simple and succinct way to specify security protocols: one only needs to describe what messages are exchanged between the protocol agents in an unattacked protocol run. However, it has turned out to be surprisingly subtle to define a formal semantics for such a notation, i.e., defining an inference system for how agents should compose, decompose and check the messages they send and receive. Such a semantics is necessary in order to automatically generate formal models and implementations from Alice-and-Bob specifications. However, even when modeling messages in the free algebra, defining the semantics has proved far from trivial [BBD+05, BN07, CVB06, JRV00, Low97a, Mil97].

To make matters worse, many modern protocols rely, for instance, on the Diffie-Hellman key agreement where the algebraic properties of modular exponentiation are necessarily part of the operational semantics, since the key exchange would be non-executable in the free algebra. For practical purposes, one can augment the semantics with support for just this special example like [Mod14], but a general and mathematically succinct and rigorous theory is desirable.

In this work, we introduce a semantics for an arbitrary set of operators and their algebraic properties. Despite this generality, the semantics is a much more succinct and mathematically simple definition than in all the previous work (it fits on half a page) because it is based on a few general and uniform principles to define the behavior of the participants. This semantics was inspired by the similar work of [Möd09, CR10], which we further simplify considerably. Our semantics is also subsuming the previous works in the free algebra and limited algebraic reasoning, as they are instances of our semantics for a particular choice of operators and algebraic properties (although this is not easy to show as explained below). We thus see our semantics as one of our main contributions since, from a mathematical point of view, a simple general principle that subsumes the complex definitions of many special cases is the most desirable property of a definition.

Our simple mathematical semantics, however, cannot be directly used as a translator from Alice-and-Bob notation to formal models or implementations since it entails an infinite representation and several of the underlying algebraic problems are in fact not recursive in general. We thus consider a particular set of operators and their algebraic properties that supports a large class of protocols, including modular exponentiation and multiplication. Our considered theory not only subsumes the theories of previous works, but also clarifies subtle details of the behavior of operators that were left implicit previously. For this theory, we define a *low-level* semantics that is much more complex than the mathematical *high-level* one but it is computable, and we formally prove that the low-level semantics is a correct implementation of the high-level one. The division into a simple mathematical high-level semantics as a "gold standard" and a low-level "implementable" semantics not only allows for a reasonable correctness criterion of the low-level semantics, but also it is in our opinion a major advantage over previous works that are a blending between mathematical and technical aspects.

To make our work applicable in practice, we have designed the *Security Protocol Specification language SPS* as a variant of existing Alice-and-Bob languages that contains many novel features that are valuable in practice. In particular, our notion of *formats* allows us to integrate the particular way of structuring messages of real-world protocols like TLS, rather than academic toy implementations; at the same time, we can use a sound abstraction of these formats in the

formal verification. We have implemented the low-level semantics in a translator that can generate both formal models in the input languages of popular security protocol analysis tools, (e.g., Applied π calculus for ProVerif [Bla01, BS11] or AVISPA IF for AVANTSSAR [AAA+12]) and implementations in JavaScript for the execution environment of the FutureID project (www.futureid.eu). We have demonstrated practical feasibility with a number of major and minor case studies, including TLS and the EAC/PACE protocols used in the German eID card.

Part of the SPS compiler is the *APCC: Automatic Protocol Composition Checker* that implements our relative soundness results. Relative soundness results, in general, have proved helpful in the automated verification of security protocols as they allow for the reduction of a complex verification problem into a simpler one, if the protocol in question satisfies sufficient conditions. These conditions are of a syntactic nature, i.e., can be established without an exploration of the state space of the protocol.

A first kind of relative soundness results are *typing results* [HLS03, BP05, Möd12a, AD14]. In this work, we consider a *typed model*: a restriction of the standard protocol model in which honest agents do not accept any ill-typed messages. This may seem unreasonable at first sight, since in the real-world agents have no way to tell the type of a random bitstring, let alone distinguish it from the result of a cryptographic operation; yet in the model, they "magically" accept only well-typed messages. The relative soundness of such a typed model means that if the protocol has an attack, then it also has a well-typed attack. This does not mean that the intruder cannot send ill-typed messages, but rather that this does not give him any advantage as he could perform a "similar" attack with only well-typed messages. Thus, if we are able to verify that a protocol is secure in the typed model, then it is secure also in an untyped model. Typically, the conditions sufficient to achieve such a result are that all composed message patterns of the protocol have a different (intended) type that can somehow be distinguished, e.g., by a tag. The restriction to a typed model, in some cases, yields a decidable verification problem, allows for the application of more tools and often significantly reduces verification time in practice [BP05, AC04].

Another kind of relative soundness results appears in *compositional reasoning*. Here, we consider the *parallel composition* of protocols, i.e., running two protocols over the same communication medium, and these protocols may use, e.g., the same long-term public keys. (In the case of disjoint cryptographic material, compositional reasoning is relatively straightforward.) The compositionality result means to show that if two protocols satisfy their security goals in isolation, then their parallel composition is secure, provided that the protocols meet certain sufficient conditions. Thus, it suffices to verify the protocols in isolation.

The sufficient conditions in this case are similar to the typing result: every composed message can be uniquely attributed to one of the two protocols, which again may be achieved, e.g., by tags.

The contributions of the second part of the thesis are twofold. First, we unify and thereby simplify existing typing and compositionality results: we recast them as an instance of the same basic principle and of the same proof technique. In brief, this technique is to reduce the search for attacks to solving constraints in a symbolic model. For protocols that satisfy the respective sufficient conditions, constraint reduction will never make an ill-typed substitution, while for compositionality "ill-typed" means to unify messages from two different protocols.

Second, this systematic approach also allows us to significantly generalize existing results to a larger set of protocols and security properties. For what concerns protocols, our soundness results do not require a particular fixed tagging scheme like most previous works, but use more liberal requirements that are satisfied by many existing real-world protocols like TLS.

While many existing results are limited to simple secrecy goals, we prove our results for the entire geometric fragment suggested by Guttman [Gut14]. We even augment this fragment with the ability to directly refer to the intruder knowledge in the antecedent of goals; while this does not increase expressiveness, it is very convenient in specifications. In fact, handling the geometric fragment also constitutes a slight generalization of existing constraint-reduction approaches.

Synopsis

This thesis is organized in two parts. The first part is "Protocol Specification, Implementation and Verification"; in this first part we proceed as follows: we define the syntax and semantics of SPS in Chapters 2 and 3 respectively. Then we discuss the connections from SPS to implementations and to formal models in Chapter 4. We give case-study protocols in Chapter 5, and discuss the related work of the first part in Chapter 6.

In the second part of the thesis titled "Protocol Composition", we introduce a symbolic protocol model based on strands in Chapters 7 and 8. Then we define security properties in the geometric fragment and how to reduce their verification to solving constraints. In Chapter 9, we give our typing and parallel compositionality results, we also introduce *APCC*: a tool that checks if protocols are parallel-composable. We discuss the related work of the second part in

Chapter 10. Finally, we conclude the thesis Chapter 11.

Part I

Protocol Specification, Implementation and Verification

Chapter 2

SPS Syntax

In this chapter we introduce the syntax of SPS using a running example that illustrates the main features of SPS. A protocol specification in SPS consists of several sections. Each section describes an aspect of the protocol, for example in the `Goals` section we specify the goals a protocol is supposed to achieve, while in the `Actions` section we specify the actions that agents perform in a protocol, e.g., the messages that the agents exchange. This chapter proceeds as follows: in Section 2.1 we give the running example and the grammar of SPS. In Section 2.2 we list of predefined types and operators of SPS and how the user can customize them. We give more details about the channels that we have in SPS in Section 2.3. Section 2.4 includes the context-sensitive properties of SPS. We introduce the *operational strands* in Section 2.5 being the target language upon which we define the semantics of SPS. In Section 2.6 we explain the preprocessing steps that we perform on SPS specifications before translating them to operational strands. We conclude this chapter in Section 2.7.

2.1 Example and Grammar

In our running example, shown in Listing 2.1, two agents `A` and `B` use a symmetric key `shk(A,B)` to establish a fresh Diffie-Hellman key and securely exchange a `Payload` message.

```
Protocol: example
Types:
    Agent A,B;
    Number g, Payload, X, Y;
Mappings:
    shk: Agent,Agent -> SymmetricKey;
Formats:
    f1(Agent, Agent, Msg);
    f2(Number);
Knowledge:
    A: A, B, shk(A,B), g;
    B: A, B, shk(A,B), g;
Actions:
    A : Number X
    A -> B  : scrypt(shk(A,B), f1(A,B,exp(g,X)))
    B : Number Y
    B -> A  : scrypt(shk(A,B), f1(B,A,exp(g,Y)))
    A : Number Payload
    A -> B  : scrypt(exp(exp(g,Y),X), f2(Payload))
Goals:
    Payload secret of A,B
```

Listing 2.1: Example Protocol in SPS

Now we give an overview of each section that appears in the SPS specification of the running example shown in Listing 2.1. Later, we explain all the sections of an SPS specification (including the ones that appear in the example).

1. `Protocol` section: the user gives a name to the specified protocol.
2. `Types` section: the user declares the identifiers of the protocol, and associates them with types, i.e., agents, symmetric keys, etc..
3. `Mappings` section: the user declares the mappings that we use in a protocol. A mapping is used to relate different protocol objects to each other e.g., an agent to its public key, or a public key to its private key.
4. `Formats` section: the user specifies the structure of plain data.
5. `Knowledge` section: the user shows the initial knowledge of each *participant* of the protocol.

6. `Actions` section: the user specifies exchanged messages and fresh data in an ideal run of the protocol.
7. `Goals` section: the user specifies the goals that the protocol is supposed to achieve, e.g., the secrecy of a message.

We give the syntax of SPS in EBNF, where we set all meta-symbols in blue and write X*s* (for a non-terminal symbol X) to denote a comma-separated list X(, X)* of X elements; Const and Func are alphanumeric strings starting with a lower-case letter (e.g., `g` and `scrypt` in the example) and Var is an alphanumeric string starting with an upper-case letter (e.g., `X` in the example).

SPS ::= [`Protocol` : Ident]
`Types` : (Type Ident*s*;)*
[`Mappings` : (Func : Type*s* → Type;)*]
[`Formats` : (Func(Type*s*);)*]
[`Macros` : (Msg = Msg;)*]
`Knowledge` : (Role : Msg*s*;)*
[`where` Role ≠ Role (& Role ≠ Role)*]
`Actions` : (Role Channel Role : Msg
| Role : Type Var
| `let` Ident = Msg)*
`Goals` : (Role `authenticates` Role `on` Msg
| Msg `secret of` Role*s*)*
[`Private` : Msg*s*]

Msg ::= Const | Var | Func(Msg*s*)

Ident ::= Const | Var | Func

Role ::= Const | Var

Type ::= `Agent` | `Number` | `PublicKey` | `PrivateKey`
| `SymmetricKey` | `Bool` | `Msg`

Channel ::= [•] → [•]

We begin our explanation with the atomic elements: constants (Const) and variables (Var). One may think of the variables as parameters of a protocol

description that must be instantiated for a concrete execution of the protocol; in our example, the variables `A` and `B` shall be instantiated with concrete agent names such as `a`, `b` or the intruder `i`, whereas `X` and `Y` should be instantiated with random numbers that are freshly chosen by `A` and `B`, respectively. Now we explain the SPS specification section-by-section.

`Protocol` Section:

This section gives a name to the protocol. The name of our example protocol is `example`. As it has no semantical significance, it is an optional section, i.e., the user may skip naming his protocol.

`Types` Section:

In the `Types` section, all constants and variables are declared with one of the pre-defined types, where the type `Msg` subsumes all types. By default, the interpretation of SPS is *untyped*, i.e., types are used only by the SPS translator to check that the user did not specify any ill-typed terms. The types can however be used to generate a more restrictive typed model and under certain conditions this restriction is without loss of attacks as we show in the second part of this thesis. The type `Agent` has a special relevance: we call the variables of this type *roles*, and the symbol ROLE in the above grammar must only be used for identifiers of type `Agent`. (This is an additional check that we cannot directly express in a context-free grammar.) A proper instantiation of roles must guarantee that each role can be played by any agent (including the intruder); we give more details on how we achieve this instantiation in Section 4.2.

While the semantics of Alice-and-Bob style languages that we give in the next section is generic for an arbitrary set of function symbols and their algebraic properties, the concrete implementation of SPS is for a set of fixed cryptographic function symbols. These are asymmetric and symmetric encryption functions (`crypt` and `scrypt`), digital signatures (`sign`), hash and keyed-hash functions (`hash` and `mac`), and modular exponentiation (`exp`) and multiplication (`mult`). There are of course corresponding operations for decryption and verification, but these are not part of an SPS specification; instead, their use is *derived* by the SPS translator according to the semantics in the next section.

`Mappings` Section:

In the `Mappings` section, one can specify a special kind of function symbols. These do not represent any actual operation that honest agents or the intruder can perform, but are used to describe the pre-existing setup of long-term keys. In our example, the mapping `shk` assigns to every pair of agents a unique value of type symmetric key; this is the easiest way to define shared keys for agents—including the intruder who will then share keys `shk`(`i`, `A`) and `shk`(`A`, `i`) with every other agent `A`. Public key infrastructures can be modeled in a similar way.

`Formats` Section:

In the `Formats` section, one can specify a third kind of function symbols called *formats*. They abstractly represent how the concrete implementation structures the clear-text part of a message, such as XML-tags or explicit message-length fields. A format thus basically represents a concatenation of information, but in contrast to a plain concatenation operator as in other formal languages, the abstract format function symbols allow us to generate implementations with real-world formats such as TLS (see below). In the example, we have two formats: `f1` is used to exchange the Diffie-Hellman half-keys together with the agent names, and `f2` indicates the transmission of the `Payload` message. For simplicity, we model a payload message using a fresh random number `Payload`, representing a placeholder for an arbitrary message (depending on the concrete application); alternatively, this could be modeled using a mapping (e.g., `payload`(`A`, `B`)) that `A` knows initially and sends to `B` after the key establishment.

The three kinds of function symbols are thus: the cryptographic function symbols, the mappings and the formats. Except for the mappings, these are all *public*: all agents, including the intruder, can apply them to messages they know. Additionally, formats are *transparent*: every agent can extract the fields of a format. We can now build composed messages with these function symbols, where we assume the additional check that all SPS messages are well-typed (and are used with the proper arity). We discuss the details of the type expressions in the “Protocol Composition” part of this thesis (Chapters 7—9).

`Macros` Section:

To keep our running example concise and short, we have not include a `Macros` section to it. However, in the case of lengthy protocol specifications, this section is very useful as it improves readability and abbreviation. For example, suppose we added to our example this code:

```
Macros:
aMacro(A1,A2,N)=scrypt(shk(A1,A2), f1(exp(g,N)))
```

Then we can replace the line:

```
A -> B  : scrypt(shk(A,B), f1(A,B,exp(g,X)))
```

with an abbreviated version of it as follows:

```
A -> B  : aMacro(A, B, X)
```

This is also applicable to the next line where `B` $\rightarrow$ `A` and the abbreviation becomes more obvious with lengthy specifications like the protocols that we specify in our case studies (Chapter 5.2).

To handle the macros that appear in a message, we simply replace the message with a macro with a new one in which we "unfold" all macros. This is so because macros have no effect on the semantics of the specification, i.e., the two lines of code that we presented above are semantically equivalent.

Note that in a macro declaration, variables on the right hand side must appear on the left hand side of the equal sign (recall that upper-case identifiers represent variables in SPS). Another way to provide code abbreviation is the `let` statement that we explain in the `Actions` section (as they occur there). Finally, note that we have another type of macros that will appear in the *operational strands*: the target language that we translate SPS to and that we discuss later in this chapter. Accordingly, from now on, we call this type of macros "syntax-macros", while the other type that we introduce later will be called strand-macros or shortly macros.

`Knowledge` Section:

In the `Knowledge` section, we specify the *initial knowledge* of each of the protocol roles. This is essential as it determines how (and if) honest agents can execute the protocol. For instance, if in the example we were to omit the item $\texttt{shk}(\texttt{A},\texttt{B})$ in the knowledge of role `B`, then `B` could not decrypt the first message from `A` and thus not obtain `A`'s half key. Moreover, in the next step, `B` could not build the response message for `A`. Also, as we will define below, this specification indirectly determines the initial knowledge of the intruder: if a role is instantiated with i, then the intruder obtains the corresponding knowledge (in our case, all shared keys $\texttt{shk}(\texttt{A},\texttt{B})$ where $\texttt{A} = \mathsf{i}$ or $\texttt{B} = \mathsf{i}$). We require that all variables in the knowledge section be of type `Agent`. Finally, one can optionally forbid some instantiations of the roles, e.g., by the side condition $\texttt{A} \neq \mathsf{i}$ or $\texttt{A} \neq \texttt{B}$.

`Actions` Section:

The `Actions` section is the core of the specification as in it we specify the messages that are exchanged between the roles of a protocol. In addition to message exchanges, we also specify freshly created values and in-line abbreviations as follows:

1. **Fresh values:** we specify here explicitly when agents freshly create new values. In our example, `A` first creates the secret exponent `X` for the Diffie-Hellman exchange, computes the half-key $\texttt{exp}(\texttt{g},\texttt{X})$, inserts it into format `f1` and encrypts the message with the shared key $\texttt{shk}(\texttt{A},\texttt{B})$ to compose the message $\texttt{scrypt}(\texttt{shk}(\texttt{A},\texttt{B}),\texttt{f1}(\texttt{A},\texttt{B},\texttt{exp}(\texttt{g},\texttt{X})))$.

2. **Message exchange:** In the exchange, a token-passing order must be followed, i.e., the receiver of the previous message is the sender of the next one.[1] In reference to our example, in order to send the message $\texttt{scrypt}(\texttt{shk}(\texttt{A},\texttt{B}),\texttt{f1}(\texttt{A},\texttt{B},\texttt{exp}(\texttt{g},\texttt{X})))$, `A` uses the standard *insecure channel* (denoted with $\rightarrow$) on which the intruder can read, intercept, and insert messages arbitrarily. SPS also supports a notion of authentic, confidential, and secure channels as in [Möd09], denoted with $\bullet\!\!\rightarrow$, $\rightarrow\!\!\bullet$ and $\bullet\!\!\rightarrow\!\!\bullet$, respectively. For instance, one may specify the exchange of the half-keys using authentic channels (without the encryption) where the intruder can see messages, but not insert messages except under his real name. This represents the *assumption* that the messages between `A` and `B` cannot be

[1] Given a protocol that initially does not follow the token-passing order, we can rewrite it as is standard, e.g., adding dummy messages to ensure this type of message flow.

manipulated by an intruder, e.g., in device pairing of mobile devices, when `A` and `B` meet physically in a public place. The assumptions are reflected only in the formal model (by restricting the intruder behavior on such channels), while in the implementation it is the duty of the surrounding software module to connect a properly secured channel to the protocol module.

3. **In-line abbreviation:** The `let` statement is the second way to provide code abbreviation in addition to the `Macros` section, we call this kind of abbreviation *in-line*. To improve specification readability in SPS and prevent code repetition, we support two syntactical ways that the user may use to abbreviate lengthy messages and reuse repeated message patterns. The second way to support specification readability in SPS is the use of `let` statements in the `Actions` section. A `let` statement is a non-parameterized abbreviation method (i.e., has no argument as opposed to macros). It is placed in the `Actions` section and applicable to all actions that follows it, i.e., it provides a local abbreviation and not applicable to the whole specification in contrast to the macros.

One last point about the `Actions` section is that it shows the simplicity of an SPS specification, i.e., this section is very similar to the way one would informally describe a protocol in Alice and Bob notation.

`Goals` Section:

In the `Goals` section, we specify the *goals* the protocol aims to achieve. SPS provides built-in goals for the standard secrecy and authentication goals. In general, we instrument the description with events that reflect what is happening during the protocol execution, e.g., the event $\mathsf{secret}(\mathtt{A}, \mathtt{B}, \mathtt{Payload})$[2] reflects that `Payload` is supposed to be a secret between `A` and `B`. We then define attack states as predicates over these events. The events allow us to formulate security goals in a protocol-independent way rather than referring to the messages of the protocol.

[2]Strictly speaking, we should write $\mathsf{event}(\mathsf{secret}(\mathtt{A}, \mathtt{B}, \mathtt{Payload}))$ but, for readability, here and below we will omit the outer $\mathsf{event}(\cdot)$ when it is clear from context.

`Private` Section:

In the final `Private` section, we specify the long-term secrets of a protocol, e.g., private keys. This section is mainly for the compositional reasoning of protocols that we discuss in more details in Chapters 7—9.

2.2 Operators and Types

Now we list all the predefined operators and types in SPS language. Those are considered as keywords in SPS.

Operators

The list of the predefined operators is:

1. Functions (public operators):
   - `scrypt`$(\cdot,\cdot)$ for symmetric key encryption.
   - `crypt`$(\cdot,\cdot)$ for asymmetric key encryption.
   - `sign`$(\cdot,\cdot)$ for signing messages.
   - `exp`$(\cdot,\cdot)$ for modular exponentiation, we omit the modulus for ease of notation.
   - `mult`$(\cdot,\cdot)$ modular multiplication, we also omit the modulus for ease of notation.
   - `hash`$(\cdot)$ for message hashing.
   - `mac`$(\cdot,\cdot)$ for message authentication code.

   Note that we do not specify the decryption and verifying operators here (e.g., the symmetric decryption operator or the signature verifier operator) as those are not allowed to appear in the SPS specification. Instead, the main task of the SPS semantics is to find them. The user is only supposed to specify the exchanged message between different agents in a protocol, then the actual action of message decomposition/decryption, checking is defined by the semantics as we show later in Chapter 3.

2. Mappings (private operators):

- `pk`: Maps an agent to a public key. For example $\texttt{pk}(A)$ gives the public key of the agent A.
- `inv`: Maps a public key to its private key. $\texttt{inv}(\texttt{pk}(A))$ gives the private key of A.
- `shk`: Maps two agents to a relation of both, it can be used to represent a key shared between them.

Note that the use of mappings allows us to systematically generate proper instantiations for formal models as we explain in Chapter 4. Finally, we would like to point out that we allow users to add their own mappings, formats and one-way functions, by simply adding them to the corresponding sections as we discussed earlier in this chapter.

Types

The predefined data types of SPS are as follows:

1. `Agent`: Used to declare agents. AGENT type is special in the sense that a special constant this type is `i` and represents the intruder. All other constants of this type (called *honest agents* hereafter) represent honest agents, and they cannot be instantiated with any other agent including the intruder `i`. Variables of this type (called *roles* hereafter) can be either honest or not; and can be instantiated with the agent `i` or any other concrete agent.
2. `PublicKey`: Used to declare public keys.
3. `Number`: Used to declare number identifiers such as an exponent or a group element.
4. `Nonce`: Used to declare random numbers.
5. `SymmetricKey`: Used to declare symmetric keys.
6. `Msg`: A generic data type used to in the untyped model. It represents the data type of composed messages, i.e., the data-type of $\texttt{sign}(k, m)$ in the untyped model is `Msg`. However, in the typed model, the type of $\texttt{sign}(k, m)$ is $\texttt{sign}(\texttt{PrivateKey}, \texttt{Number})$ given that k is of type `PrivateKey` and m is of type `Number`. The details of the typed model will be given in Chapter 7.

Customization of SPS operators and types

As we mentioned earlier, the primary goal of SPS is to generate implementations and formal models for real-world protocols and that both generated results coincide. In order to generate more realistic implementations from SPS specifications, SPS allows the user to customize the operators and types. With customization we mean the ability to specify attributes of the operator or type, e.g., the size of the type `Number` or the algorithm used for the symmetric encryption `scrypt`. A user can customize an operator (or a type) simply by adding an annotation to it in the style of a comma-separated list of name/value pairs. In each of these pairs, the name must be a string, while the value can be numeric or a term, a string enclosed in double quotation marks, or a term that was previously used in the SPS specification. Consider the following code examples:

- $\texttt{A} : \texttt{SymmetricKey}[\texttt{size} = 2048]\ K$
  means that the agent `A` creates a symmetric key K that has the size of 2048 bits.

- $\texttt{A} \rightarrow \texttt{B} : \texttt{p1}, \texttt{p2}$
  $\texttt{B} \rightarrow \texttt{A} : \texttt{crypt}[\texttt{algorithm} = \texttt{p1}, \texttt{keysize} = \texttt{p2}](\texttt{k}, \texttt{Hi})$
  means that `A` first sends to `B` the two parameters `p1` and `p2`, then `B` sends to `A` the message `Hi` encrypted with the public key `k` using the encryption algorithm and the keysize specified in `p1` and `p2` respectively. This can be seen in the context of a negotiation stage of a protocol, in which one participant sends to the other his preferences and the other comply with.

Note that this customization affects only the generated implementation and not the formal model; we think that the elimination of such fine details is an accepted over-approximation at least in the symbolic model.

2.3 Channels

In SPS, we distinguish between several types of channels:

1. Insecure channel: $A \rightarrow B : M$ represents the default insecure channel from A to B, controlled by the intruder. Intruder can read, send under any sender's name, and intercept messages.

2. Authentic channel: $A \bullet\!\!\rightarrow B : M$ represents an authentic channel from A to B. Here B is guaranteed that the message is sent from A (and meant for B). However, the intruder can see the message M.

3. Confidential channel: $A \rightarrow\!\!\bullet\, B : M$ means that A has the guarantee that only the intended receiver B can see the message M. However, B has no guarantee of authenticity.

4. Secure Channel: $A \bullet\!\!\rightarrow\!\!\bullet\, B : M$ represents an authentic and confidential channel.

5. Pseudonymous channels: $[A]_\psi \bullet\!\!\rightarrow\!\!\bullet\, B : M$ and $B \bullet\!\!\rightarrow\!\!\bullet\, [A]_\psi : M$. This represents a secure channel, but with an unauthenticated party A that acts under pseudonym ψ. This is different from a channel where the endpoint A is simply not secured, i.e., $A \rightarrow\!\!\bullet\, B$ or $B \bullet\!\!\rightarrow A$, because the channel is bound to pseudonym ψ. The idea is to model channels like the ones we get from TLS without client authentication: we have a secure line between a client and a server, but the identity of the client is not proved. However, an intruder cannot hack into this line any more. This is crucial when the client uses the channel for a login to authenticate itself. We also allow to drop the notation ψ of the pseudonym if not relevant for the protocol: it then means that at the beginning of each protocol run, each pseudonymous user picks a fresh pseudonym to use throughout the session.

6. Mutual pseudonymous channel $[A]_\psi \bullet\!\!\rightarrow\!\!\bullet\, [B]_\varphi : M$ represents a secure channel with both parties unauthenticated to each other. Each participant is under a pseudonym (A under ψ and B under φ that can be both dropped as in the previous channel type).

2.4 Context-Sensitive Properties

Now we present some of the properties of SPS syntax that we did not express in the grammar previously. SPS is case-sensitive and any line that starts with # is a comment. More precisely, whatever lies between a # and the end of the line is a comment. The only variables that may appear in the initial knowledge of any agent (the `Knowledge` section) must be of type `Agent`. The identifiers cannot be reused, e.g., an identifier defined as a constant cannot be re-defined as a format. The arity of a user-defined function is derived from the first occurrence of that function, i.e., if the user declares a function in the `Types` Section, and he uses it with four arguments in the first occurrence, then it must be consistently used with four arguments. The arity of a format is the number of fields in its declaration. The arity of pre-defined operators is fixed, e.g., `scrypt` and `exp` take two arguments. In case the user wants to encrypt a message of

three fields, then he needs to use a format. i.e., instead of $\texttt{crypt}(k, m_1, m_2)$ use $\texttt{crypt}(k, \texttt{f}_2(m_1, m_2))$ where $\texttt{f}_2$ is a format with two arguments. Formats, mappings and functions must be used with the proper arity, e.g., a format declared with three arguments must be always given three arguments. The atomic messages that appear on the right-hand side of a macro declaration must be bound, i.e., they must be either declared previously in the `Types` section or occur on the left-hand side of the macro declaration. For example, the declaration $aMacro(X, Y) = \texttt{f}_1(A, B, c, X, Y)$ is not accepted by the SPS compiler unless A, B and c are previously declared in the `Types` section.

2.5 Operational Strands

As a preparation for defining the SPS semantics, we first define the target language that we call *operational strands*. We define operational strands as an extension of the popular *strands* [THG99]. In the original definition of [THG99], a strand denotes a part of a concrete protocol execution, namely, a sequence of ground messages that an agent sends and receives. Our operational strands extends the strands of [THG99] with several notions that we explain shortly.

The Syntax of Operational Strands

The syntax of operational strands is a slight extension of the well-known strand spaces:

STRAND ::= KNOWLEDGE: (send(CHANNEL, MSG). | receive(CHANNEL, MSG).
| event(MSG). | MSG $\doteq$ MSG. | VAR:=MSG. | fresh VAR.)* 0

The non-terminals CHANNEL, MSG, and VAR are as in the SPS syntax. KNOWLEDGE, typically denoted by M in concrete strands, stands for a knowledge as defined in Definition 3.2, i.e., a substitution from label variables to protocol terms. We may omit this knowledge prefix of an operational strand when not relevant, as it is mainly used as an annotation in the semantics of SPS. Moreover, operational strands may include equations on messages that we discuss later in Section 8.2.

Finally, we have a restricted version of operational strands that we call *plain strands*. Each agent of a protocol has his own plain strand that shows how the

protocol looks like from the point of view of that agent in an ideal protocol run: what messages it is supposed to send and what messages it receives. Unlike operational strands, plain strands do not give the exact details of how messages should be derived or checked. Syntactically speaking, plain strands do not include equalities. Moreover, the set of variables that occur in plain strands are disjoint from the ones that appear in operational strands, but these details will be given later. Plain strands are the result of the first step towards the definition of our operational semantics. This step is simply splitting the SPS specification into a strand for each agent. From now on we say *plain strand* when we strictly refer to one of the non-detailed strands, otherwise we use strand or operational strand to refer to the full version of operational strands. A concrete example of a plain strand is shown Figure 2.2, and the splitting step is indicated by the dotted line in Figure 2.1. Furthermore, we give in Figure 3.1 the plain and operational strands of the agent `A` of our example protocol. The goal of SPS semantics is to translate plain strands to operational strands; we give the formal details of this translation in and operational strands in Chapter 3.

We extend the strands of Guttman [THG99] as follows:

First, send and receive steps can be annotated with a channel. Recall that SPS supports default insecure channels as well as authentic, confidential and secure ones. For the SPS semantics, this is only a label on the channels that is left unchanged in the translation; for the semantics of operational strands, the channels mean a restriction on the operations that the intruder can perform on the channel as explained in Section 3.5. In the following, we use a textual representation for strands for simplicity and brevity. We write $\mathsf{send}(ch, t)$ and $\mathsf{receive}(ch, t)$ for sending and receiving message t over channel ch. For example the plain strand for `A` in Figure 2.2 would be written as: (Let $M^{\mathtt{A}}$ be the initial knowledge of A.)

$M^{\mathtt{A}}$: $\mathsf{fresh}\ \mathtt{X}.\mathsf{send}(ch, \mathtt{scrypt}(\mathtt{shk}(\mathtt{A},\mathtt{B}), \mathtt{f}_1(\mathtt{A},\mathtt{B},\mathsf{exp}(\mathtt{g},\mathtt{X})))).$
$\mathsf{receive}(ch, \mathtt{scrypt}(\mathtt{shk}(\mathtt{A},\mathtt{B}), \mathtt{f}_1(\mathtt{B},\mathtt{A},\mathsf{exp}(\mathtt{g},\mathtt{Y})))).$
$\mathsf{fresh}\ \mathtt{Payload}.\mathsf{send}(ch, \mathtt{scrypt}(\mathsf{exp}(\mathsf{exp}(\mathtt{g},\mathtt{X}),\mathtt{Y}), \mathtt{f}_2(\mathtt{Payload})).$
$\mathsf{event}(\mathsf{secret}(\mathtt{A},\mathtt{B},\mathtt{Payload})))$

Second, we annotate each strand with the initial knowledge of the role it represents, denoted by a box above the strand (we define knowledge formally in Definition 3.2). The annotation has no meaning for the behavior of strands and is only needed during the translation process. In textual representation, we write the annotation with the knowledge M as M : *steps* at the beginning of the strand as we shown in the previous example.

Third, recall that the original strand spaces are used to characterize sets of protocol executions and contain only ground terms. In contrast, we use them like a "light-weight" process calculus: terms may contain variables (representing

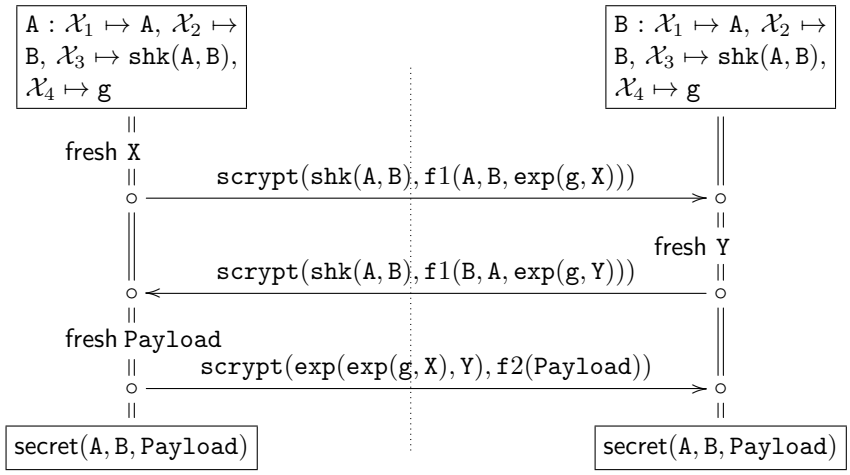


Figure 2.1: The Example Protocol as a Message Sequence Chart

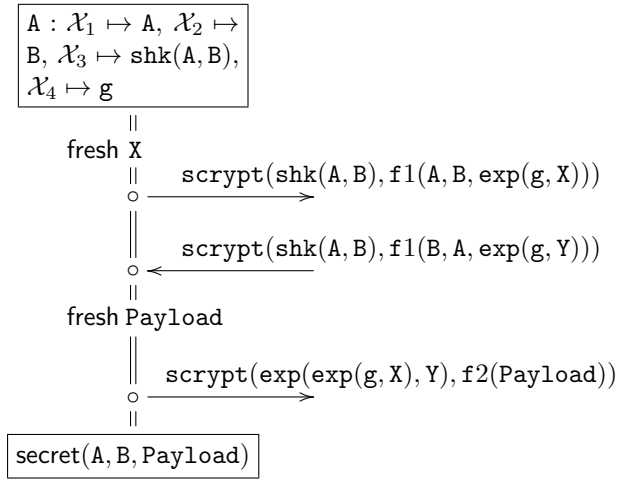


Figure 2.2: The Plain Strand of A

values that are instantiated during the concrete execution). Also, we have the construct fresh X where the variable X will be bound to a fresh value. An important requirement is that operational strands are *closed* in the following sense: every variable must be *bound* by first occurring in the initial knowledge, in a fresh operation, in a *strand-macro* (that we introduce shortly), or in a receive step. A bound variable must not occur subsequently in a fresh operation (i.e., it cannot be "re-bound"). In contrast, a bound variable may occur in a subsequent receive step, meaning simply that the agent expects the same value that the variable was bound to before.

Fourth, we extend strands with *events* (predicates over terms) to formulate security goals in a protocol-independent way. For instance, as we already remarked above, we may use the event secret(A, B, Payload) to express that message Payload is regarded as a secret between protocol roles A and B. Then we can define (independent of the concrete protocol) a violation of secrecy as a state where the intruder has learned Payload but is neither A nor B. We do not give here more details on goals, because from a semantical point of view we just treat the events as if they were messages on a special channel to a "referee" who decides if the present state is an attack; the handling of these events is uniform for a wide class of goals (explained in the second part of this thesis) and only limited by the abilities of current verification tools. In textual representation, we will simply write event(t) where t is a term characterizing the event.

Fifth, we add *checks* of the form $s \doteq t$. The meaning is that the agent can only continue if the terms s and t are equal and aborts otherwise. Also, we have *strand-macros* of the form $\mathcal{X}_i := t$, which mean that we consider the same strand with all occurrences of $\mathcal{X}_i$ replaced by t. This is helpful for generating protocol implementations, because the result of a computation t is stored in a variable $\mathcal{X}_i$ and does not need to be computed again later. Note that strand-macros are different from the syntax-macros that we presented in the SPS specification (in Macros section), the latter are abbreviations that occur in the SPS specifications while the strand-macros are the ones that occur in the strands. From now on, we use the term macros to refer to the strand-macros.

We define the free variables of an operational strand as follows:

$$
\begin{aligned}
fv(M : steps) &= fv(steps) \setminus dom(M) \\
fv(\mathsf{send}(ch, t).rest) &= fv(ch) \cup fv(t) \cup fv(rest) \\
fv(\mathsf{receive}(ch, t).rest) &= (fv(rest) \setminus fv(t)) \cup fv(ch) \\
fv(\mathsf{event}(t).rest) &= fv(t) \cup fv(rest) \\
fv(s \doteq t.rest) &= fv(s) \cup fv(t) \cup fv(rest) \\
fv(x := t.rest) &= (fv(rest) \setminus \{x\}) \cup fv(t) \\
fv(\mathsf{fresh}\ x.rest) &= fv(rest) \setminus \{x\} \\
fv(0) &= \emptyset \\
fv(x) &= \{x\} \\
fv(f(t_1, \ldots, t_n)) &= fv(t_1) \cup \ldots \cup fv(t_n)
\end{aligned}
$$

We require that all operational strands are *closed*, i.e., all variables, before being "used", are bound by occurring in the knowledge, in a received message, or in a fresh step. Further, a bound variable cannot occur in a fresh step (e.g., fresh x.fresh x.0 is not allowed) or a macro (e.g., $x := x$ cannot occur in a strand, since then x is bound earlier, violating that it cannot be re-bound, or x is a free variable of the strand). When a bound variable occurs in a receive step, it is not "re-bound", i.e., $\mathsf{receive}(ch, x).\mathsf{receive}(ch, x).rest$ by the following semantics will be equivalent to $\mathsf{receive}(ch, x).\mathsf{receive}(ch, y).x \doteq y.rest$. A formal definition of operational strands can be given as a process (interacting with a given environment). We define the semantics of operational strands at the end of the next chapter (namely in Section 3.5) as it relies on some definitions that we need to give first.

2.6 Preprocessing of SPS Specification

In the next chapter (Chapter 3), we define the semantics of SPS by a translation to operational strands. Before that translation can take place in the compiler, we need to perform a *preprocessing stage* on the SPS specification. This stage aims at removing the details that do not affect the semantics. The steps of this preprocessing stage are as follows.

1. Translation to plain strands: in this step we split the SPS specification into several plain strands (a plain strand for each participant of the protocol). This step does not define the exact actions of each participant, instead it only specifies the protocol specification from the point of view of one participant. This step is also referred elsewhere as end-point projection [CHY07, BBD+05] . In reference to our example shown in Listing 2.1

and depicted as a message sequence chart (MSC) in Figure 2.1, the result of this step is the two plain strands of the agents `A` and `B`, and we show the plain strand of `A` in Figure 2.2.

2. Type checking: we make this step optional for users who would like to apply a sanity check over the data types and their consistency throughout the specification. This step results in error messages in case of type flaws occurring in the specifications. More details about the typed model of SPS is found in Chapter 9.

3. Unfold syntax-macros and `let` statements: As both of them are just for abbreviation and have no semantical significance, we perform a preprocessing step of unfolding syntax-macros and `let` statements. i.e., we simply replace them wherever they occur in the specification with what they abbreviate.

4. Channel preprocessing: Since we support different channel types in protocol specification, we perform a channel preprocessing step in which we model channels via cryptographic operations, e.g., signing for authentic channels as in [MV09a].

2.7 Summary

In this chapter we introduced the syntax of SPS using a running example and the grammar of the language. Of the many modeling features that we have in SPS, we introduced the formats that abstractly represent how concrete implementations structure the clear-text of a message. Mappings are another modeling feature of SPS that allows for a proper instantiation of agents needed for instance in key distribution. We listed the pre-defined primitives and types, and we also specified the channels that we support in SPS. We explained the context-sensitive properties to conclude the syntax of SPS. We also presented the syntax of the operational strands, being the target language that we translate SPS to it in order to the semantics of SPS in the next chapter. Before we perform this translation, we need to run a preprocessing step on SPS specifications that we have explained as well. In the next chapter, we define the semantics of SPS.

Chapter 3

SPS Semantics

In Chapter 2 we described the SPS syntax for a fixed set of cryptographic operators (for which we give a fixed set of algebraic equations). We also described the syntax of the operational strands being the target language that we define the semantics of SPS upon. In this chapter, we give a semantics that is parameterized over an *arbitrary* set of operators and algebraic properties, inspired by [Möd09, CR10]. One of the main contributions of our work is to give this general definition of a semantics for Alice-and-Bob style languages in a concise, mathematical way that is based on a few simple, general principles shown in Sections 3.1—3.3. The semantics is a function from SPS to operational strands (via plain strands); this function is in general not recursive because many of the underlying algebraic reasoning problems are not. The value of this general definition is its simplicity and uniformity: this is in fact the best mathematical argument why to define a concept in a particular way and not differently. In Section 3.4, we then show that we can actually implement this semantics for the operators of SPS; in fact, we define a "low-level" semantics that is a computable function from SPS to operational strands and prove that it coincides with the general "high-level" semantics. In Section 3.5, we define the semantics of operational strands as an infinite transition system based on the definitions that we give in the previous sections.

3.1 Message Model

We define messages as algebraic terms and use the words *message* and *term* interchangeably. We distinguish between two kinds of messages: (1) the *protocol messages* that appear in an SPS specification and (2) *labels* (or *recipes*) that are the messages in the strands which the semantics translates to. It is necessary to make this distinction as the SPS specification reflects the ideal protocol run, while the semantics reflects the actual actions and checks that an honest agent performs in the run of the protocol. For the same reason, we will also distinguish between two kinds of variables: *protocol variables* and *label variables*.

Definition 3.1 A *message model* is a four-tuple $(\Sigma, V, \mathcal{L}, \approx)$. Σ is a countable set of *function symbols*, all denoted by lower-case letters, where: $\Sigma_0 \subseteq \Sigma$ is a countable set of *constants*, $\Sigma_p \subseteq \Sigma$ is a finite set of *public operators* such as public-key encryption, and $\Sigma_m \subseteq \Sigma$ is a finite set of *mappings* (or *private operators*), disjoint from Σ_p. We assume a global public constant $\top \in \Sigma_p \cap \Sigma_0$. V is a countable set of *protocol variables*. $\mathcal{L} = \{\mathcal{X}_1, \mathcal{X}_2, \mathcal{X}_3 \ldots\}$ is a countable set of *label variables* disjoint from Σ and V. $\approx$ is a congruence relation over *ground terms* over Σ (i.e., terms without variables), which are denoted by $\mathcal{T}_\Sigma$. A *term* is thus a constant, a variable, or an application of a function f ($f \in \Sigma$) of arity n on n terms. We write $\mathcal{T}_S(A)$ for the set of terms over a signature S and variables from set A.

As we define a deduction relation below, the public operators in Σ_p are those functions that every agent and the intruder can apply to messages they know, i.e., the cryptographic operators (including operators for decryption that do not occur in the SPS specification) and the non-cryptographic formats. In contrast, the mappings in Σ_m are private, like `shk` in our example protocol that maps from two agents to their shared secret key, or `inv` that maps from public to private keys.

Example 3.1 *As a concrete example of a message model that is representative for a large class of security protocols, let Σ_p contain all operators of the equations in Table 3.1, where $\approx$ is the least congruence relation satisfying the equations. For instance,* `scrypt` *represents symmetric encryption,* `dscrypt` *is the corresponding decryption operator and* `vscrypt` *is a* verifier*: given a term t and a key k, it tells us whether t is a valid symmetric encryption with key k. This models the fact that most symmetric ciphers include measures to detect when the decryption fails (e.g., when it is actually not an encrypted message or the given key is not correct) and in concrete implementations this verification*

Table 3.1: Example of an equational theory $\approx$

(1)	$\mathtt{dscrypt}(k,\ \mathtt{scrypt}(k,m)) \approx m$
(2)	$\mathtt{vscrypt}(k,\ \mathtt{scrypt}(k,m)) \approx \top$
(3)	$\mathtt{dcrypt}(\mathtt{inv}(k),\ \mathtt{crypt}(k,m)) \approx m$
(4)	$\mathtt{vcrypt}(\mathtt{inv}(k),\ \mathtt{crypt}(k,m)) \approx \top$
(5)	$\mathtt{open}(\mathtt{sign}(k,m)) \approx m$
(6)	$\mathtt{vsign}(k,\mathtt{sign}(\mathtt{inv}(k),m)) \approx \top$
For every $\mathtt{f} \in \Sigma_{\mathtt{f}}$ with arity n and for every $i \in \{1,\ldots,n\}$	
(7)	$\mathtt{get}_{i,\mathtt{f}}(\mathtt{f}(t_1,\ldots,t_n)) \approx t_i$
(8)	$\mathtt{verify}_{\mathtt{f}}(\mathtt{f}(t_1,\ldots,t_n)) \approx \top$
(9)	$\mathsf{exp}(\mathsf{exp}(t_1,t_2),t_3)) \approx \mathsf{exp}(t_1,\mathtt{mult}(t_2,t_3))$
(10)	$\mathtt{mult}(t_1,t_2) \approx \mathtt{mult}(t_2,t_1)$
(11)	$\mathtt{mult}(t_1,\mathtt{mult}(t_2,t_3)) \approx \mathtt{mult}(\mathtt{mult}(t_1,t_2),t_3)$

will be part of the call to dscrypt*. We emphasize that our message model explicitly describes such fine details that most security protocol analysis tools silently assume; we could similarly define a set of primitives that do not have verifiers (e.g.,* vcrypt*) and the semantics will accordingly define which verifications honest agents can and cannot do.*

Similarly, the operators crypt*,* dcrypt *and* vcrypt *formalize asymmetric encryption, and* sign*,* open *and* vsign *formalize digital signatures.*

Let $\Sigma_f \subseteq \Sigma_p$ be a set of formats declared in an SPS specification. Then, for each format $f \in \Sigma_f$ of arity n, $\mathtt{get}_{i,\mathtt{f}} \in \Sigma_p$ is an extraction function *for the i-th field of the format (for all $1 \leq i \leq n$) and $\mathtt{verify}_{\mathtt{f}} \in \Sigma_p$ is a verifier to check that a given message has format* f*.*

Moreover, we have exp *and* mult *for modular exponentiation and multiplication as needed in Diffie-Hellman-based protocols. As is often done, we omit the modulus for ease of notation. Σ_p also contains* hash *and* mac *representing hash and keyed hash functions, respectively (*hash *and* mac *do not appear in Table 3.1 since they have no algebraic properties). Finally, a typical set of mappings could be:* $\mathtt{shk} : \mathtt{Agent} \times \mathtt{Agent} \to \mathtt{SymmetricKey}$ *to denote a shared key of two agents,* $\mathtt{pk} : \mathtt{Agent} \to \mathtt{PublicKey}$ *for the public key of an agent, and* $\mathtt{inv} : \mathtt{PublicKey} \to \mathtt{PrivateKey}$ *for the private key corresponding to a given public key. Although* pk *is typically publicly available, it should not be a public operator as it does not correspond to a computation that honest agents or the intruder can perform (rather the initial distribution of keys should be specified in the knowledge section of SPS).* □

Definition 3.2 A *labeled message* t^l consists of a *protocol message* $t \in \mathcal{T}_\Sigma(V)$

and a *label* $l \in \mathcal{T}_{\Sigma_p}(\mathcal{L})$. A *knowledge* is a substitution of the form $M = [\mathcal{X}_1 \mapsto t_1, \ldots, \mathcal{X}_n \mapsto t_n]$, where $\mathcal{X}_i \in \mathcal{L}$ and $t_i \in \mathcal{T}_\Sigma(V)$. We call the set $\{\mathcal{X}_1, \ldots, \mathcal{X}_n\}$ the *domain* of M and write $|M| = n$ for the *length* of M. We may also refer to M as a set of entries and write, e.g., $M \cup \{\mathcal{X}_j \mapsto t_j\}$ to add a new entry (where $\mathcal{X}_j$ is not in the domain of M).

Intuitively, the label variables represent *memory locations* of an honest agent. A label l is composed from label variables and public operators, and reflects what actions an honest agent has performed on elements of its knowledge. A labeled message t^l expresses that an honest agent performed the actions of l to obtain what the SPS specification represents by the term t.

A knowledge $M = [\mathcal{X}_1 \mapsto t_1, \ldots, \mathcal{X}_n \mapsto t_n]$ thus represents the local state of an honest agent as a set of memory positions and corresponding SPS terms. For instance, we represent the initial knowledge of `A` in Listing 2.1 by $[\mathcal{X}_1 \mapsto \mathtt{A}$, $\mathcal{X}_2 \mapsto \mathtt{B}$, $\mathcal{X}_3 \mapsto \mathtt{shk(A,B)}$, $\mathcal{X}_4 \mapsto \mathtt{g}]$ to express that `A` stores her name and `B`'s name in her memory locations $\mathcal{X}_1$ and $\mathcal{X}_2$, a key shared with `B` in $\mathcal{X}_3$, and the group `g` in $\mathcal{X}_4$.

3.2 Message Derivation and Checking

We now define how *honest* agents can derive terms from their knowledge. This is in the style of Dolev-Yao deduction relations, but extended to labeled messages to keep track of the operations that have been applied. The relation has the form $M \vdash t^l$ where M is a knowledge and t^l a labeled term.[1]

Definition 3.3 $\vdash$ is the least relation that satisfies the following rules:

$$\frac{}{M \vdash t^{\mathcal{X}_i}} \quad Ax\,, \; [\mathcal{X}_i \mapsto t] \in M \qquad \frac{M \vdash t^l}{M \vdash s^m} \quad Eq\,, \; s \approx t, l \approx m$$

$$\frac{M \vdash t_1^{l_1} \;\ldots\; M \vdash t_n^{l_n}}{M \vdash f(t_1, \ldots, t_n)^{f(l_1,\ldots,l_n)}} \quad Cmp\,, \; f \in \Sigma_p$$

The rule Ax expresses that an agent can deduce any message that it has in its knowledge, Eq expresses that deduction is closed under equivalence in $\approx$ (on

[1] One may employ an entirely different model for the intruder (e.g., a cryptographic one); using a Dolev-Yao style deduction for honest agents is simply the semantical decision that they perform only standard public operations (that would be part of a crypto API), but no operations that would amount to cryptographic attacks.

terms and their labels), and *Cmp* allows agents to apply any public operator to deducible terms.

Example 3.2 *Consider again the algebraic theory of Table 3.1 and the knowledge* $M = [\mathcal{X}_1 \mapsto \mathtt{k}, \mathcal{X}_2 \mapsto \mathtt{X}, \mathcal{X}_3 \mapsto \mathtt{scrypt}(\mathtt{k}, \mathsf{exp}(\mathsf{g}, \mathtt{Y}))]$. *$M$ contains three messages (or "memory locations") $\mathcal{X}_1, \ldots, \mathcal{X}_3$ that we associate with the corresponding messages of the SPS specification. We explain later how to reach a particular memory state, but for the intuition let us just consider an example scenario that would produce M for an agent A: the constant k could be part of the initial knowledge of A, $\mathtt{X}$ could be her secret Diffie-Hellman exponent, and the message stored in $\mathcal{X}_3$ could be what she received from another agent—*supposedly *the Diffie-Hellman half-key* $\mathsf{exp}(\mathsf{g}, \mathtt{Y})$ *encrypted with the key k. The tricky part here is that in general A will be unable to check that the received message has the correct form (i.e., that she did not receive just some garbage); it is part of the semantics to describe what A can check and what messages she will construct on the basis of the labels $\mathcal{X}_1, \ldots, \mathcal{X}_3$. Let us for instance consider the case that A should now—according to the SPS specification—generate the Diffie-Hellman full-key $t = \mathsf{exp}(\mathsf{exp}(\mathsf{g}, \mathtt{X}), \mathtt{Y})$. That amounts to finding a label l such that $M \vdash t^l$, i.e., that* would *produce the Diffie-Hellman key, if the received message has the required form. Indeed, there is such a label as the following proof tree shows:*

$$\dfrac{\dfrac{\dfrac{}{M \vdash \mathtt{X}^{\mathcal{X}_2}}\,Ax \qquad \dfrac{\dfrac{\dfrac{}{M \vdash \mathtt{k}^{\mathcal{X}_1}}\,Ax \qquad \dfrac{}{M \vdash \mathtt{scrypt}(\mathtt{k}, \mathsf{exp}(\mathsf{g}, \mathtt{Y}))^{\mathcal{X}_3}}\,Ax}{M \vdash \mathtt{dscrypt}(\mathtt{k}, \mathtt{scrypt}(\mathtt{k}, \mathsf{exp}(\mathsf{g}, \mathtt{Y})))^{\mathtt{dscrypt}(\mathcal{X}_1, \mathcal{X}_3)}}\,Cmp}{M \vdash \mathsf{exp}(\mathsf{g}, \mathtt{Y})^{\mathtt{dscrypt}(\mathcal{X}_1, \mathcal{X}_3)}}\,Eq}{M \vdash \mathsf{exp}(\mathsf{exp}(\mathsf{g}, \mathtt{Y}), \mathtt{X})^{\mathsf{exp}(\mathtt{dscrypt}(\mathcal{X}_1, \mathcal{X}_3), \mathcal{X}_2)}}\,Cmp}{M \vdash \mathsf{exp}(\mathsf{exp}(\mathsf{g}, \mathtt{X}), \mathtt{Y})^{\mathsf{exp}(\mathtt{dscrypt}(\mathcal{X}_1, \mathcal{X}_3), \mathcal{X}_2)}}\,Eq$$

In fact, we see the "recipe" to generate the term $\mathsf{exp}(\mathsf{exp}(\mathsf{g}, \mathtt{X}), \mathtt{Y})$ *in the label* $\mathsf{exp}(\mathtt{dscrypt}(\mathcal{X}_1, \mathcal{X}_3), \mathcal{X}_2)$, *i.e., A has to first apply decryption to term $\mathcal{X}_3$ using the term $\mathcal{X}_1$ as decryption key; if the received $\mathcal{X}_3$ message was indeed of the right form, this gives the other agent's half-key (*$\mathsf{exp}(\mathsf{g}, \mathtt{Y})$ *in SPS), and this is further exponentiated with $\mathcal{X}_2$ to supposedly yield the full key (*$\mathsf{exp}(\mathsf{exp}(\mathsf{g}, \mathtt{Y}), \mathtt{X})$ *in SPS). Note that the semantics also tells us what happens if A in the actual execution receives some improper term for $\mathcal{X}_3$. In such a case, A will simply apply the operations to the term as prescribed and that may lead for instance to the protocol getting stuck (if nobody else can generate the key) or to an attack (if the intruder manages to find a term that breaks some security goals), or the garbage term may actually be detected by the checks on messages that we*

describe next, which in this example amounts to checking that the given term is indeed an encryption with the right key. □

The definition of the checks that honest agents can make on their knowledge is in fact based on the deduction relation $\vdash$. The checks will be written as equations between terms. To that end, we introduce the symbol $\doteq$ and define $\doteq$*-equations* as follows: an *interpretation* $\mathcal{I}$ is a total mapping from $\mathcal{L}$ to $\mathcal{T}_\Sigma(V)$ that we extend to a function from $\mathcal{T}_\Sigma(V \cup \mathcal{L})$ to $\mathcal{T}_\Sigma(V)$ as expected; then we define $\mathcal{I} \models s \doteq t$ iff $\mathcal{I}(s) \approx \mathcal{I}(t)$, and extend this to (finite or infinite) conjunctions of equations as expected. We define $\phi \models \psi$ iff $\mathcal{I} \models \phi$ implies $\mathcal{I} \models \psi$ for every interpretation $\mathcal{I}$; and $\phi \equiv \psi$ iff both $\phi \models \psi$ and $\psi \models \phi$.

Definition 3.4 We define a *complete set of checks* $ccs(M)$ for a knowledge M as follows: $ccs(M) = \bigwedge\{l_1 \doteq l_2 \mid \exists\, m \in \mathcal{T}_\Sigma(V).\, M \vdash m^{l_1} \wedge M \vdash m^{l_2}\}$.

$ccs(M)$ yields an infinite conjunction of checks that an agent can perform on his knowledge. Intuitively, $M \vdash m^{l_1}$ and $M \vdash m^{l_2}$ express that, according to the SPS specification, computing l_1 and l_2 *should* yield the same result m, and the agent can thus check that they actually do. For instance, consider $M = [\mathcal{X}_1 \mapsto k, \mathcal{X}_2 \mapsto \mathsf{hash}(m), \mathcal{X}_3 \mapsto \mathtt{scrypt}(k, m)]$. Amongst others, $ccs(M)$ then entails the checks $\phi = \mathsf{vscrypt}(\mathcal{X}_1, \mathcal{X}_3) \doteq \top \wedge \mathsf{hash}(\mathtt{dscrypt}(\mathcal{X}_1, \mathcal{X}_3)) \doteq \mathcal{X}_2$, i.e., the agent A can verify that $\mathcal{X}_3$ is an encryption and that $\mathcal{X}_2$ is the hash of the content of the encrypted message $\mathcal{X}_3$. Note that there are many more equations (e.g., $\mathcal{X}_1 \doteq \mathcal{X}_1$) and for every equation $s \doteq t$, we also have $h(s) \doteq h(t)$ for every unary public operator h. However, it holds that $ccs(M) \equiv \phi$, i.e., $ccs(M)$ is logically equivalent to ϕ and thus all other checks are redundant.

We will later show in the low-level model how to generally compute for our example message model such a finite set ϕ of checks equivalent to $ccs(M)$.

3.3 High-level Semantics

Now we can put everything together to define the semantics of SPS specifications by translating specifications to operational strands. Figure 2.1 shows our example protocol in the style of message sequence charts. The first step towards an operational semantics is to split the protocol into plain strands, one for each role as indicated by the dotted line in Figure 2.1. The result of applying this step on Figure 2.1 is shown in Figure 2.2. Recall that each plain strand shows how the protocol looks like from the point of view of that role in an ideal protocol run: what messages it is supposed to send and what messages it receives.

The second step towards the operational semantics is to identify the precise set of actions, i.e., how messages are composed or decomposed, and what checks need to be performed on received messages. The right part of Figure 3.1 shows how this operational description looks like for role A of the example (role B is very similar).

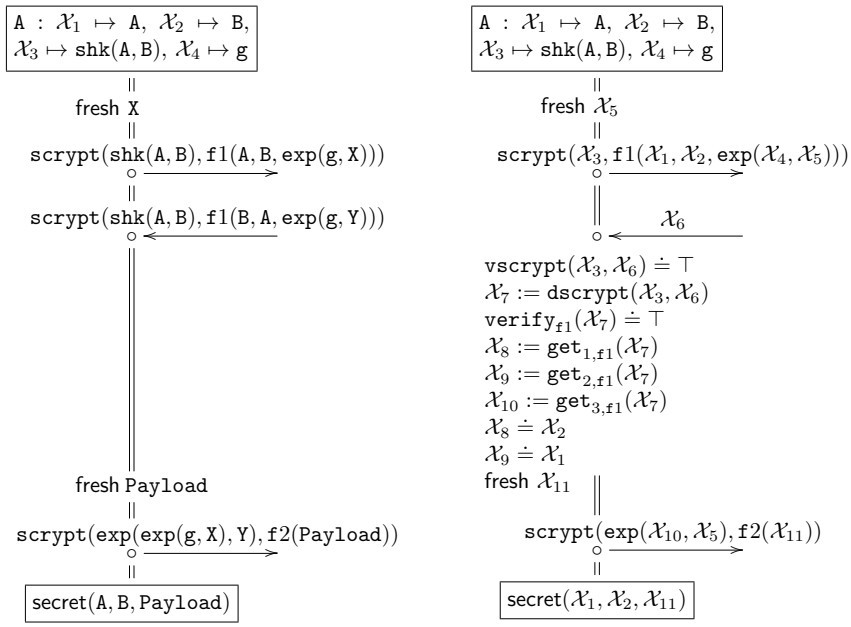


Figure 3.1: The Plain and Operational Strands of A

Now we define the *high-level semantics* as a function $[\![\cdot]\!]_H$ (with initial case $[\![\cdot]\!]_{H_0}$) that maps from plain strands like (a) to the operational strands like (b).

In a nutshell, we use the labeled deduction $M \vdash t^l$ to define how an agent composes an outgoing message (or event), and we use the *ccs* function whenever an agent receives a new message, formalizing the set of checks that the agent can perform at this point. Note that this is an infinite conjunction and we later show how to obtain an equivalent finite conjunction for the example theory.

Definition 3.5 $[\![\cdot]\!]_H$ translates from plain to operational strands as follows:

$$
\begin{array}{ll}
[\![M : steps]\!]_{H_0} & = M : ccs(M).[\![steps]\!]_H(M) \\
[\![\mathsf{receive}(ch,t).rest]\!]_H(M) & = \mathsf{receive}(ch, \mathcal{X}_{|M|+1}).ccs(M \cup [\mathcal{X}_{|M|+1} \mapsto t]). \\
 & \quad [\![rest]\!]_H(M \cup [\mathcal{X}_{|M|+1} \mapsto t]) \\
[\![\mathsf{send}(ch,t).rest]\!]_H(M) & = \mathsf{send}(ch,l).[\![rest]\!]_H(M) \\
 & \text{where } l \text{ is some label with } M \vdash t^l \\
[\![\mathsf{event}(t).rest]\!]_H(M) & = \mathsf{event}(l).[\![rest]\!]_H(M) \\
 & \text{where } l \text{ is some label with } M \vdash t^l \\
[\![\mathsf{fresh}\ X.rest]\!]_H(M) & = \mathsf{fresh}\ \mathcal{X}_{|M|+1}.[\![rest]\!]_H(M \cup \{\mathcal{X}_{|M|+1} \mapsto X\}) \\
[\![0]\!]_H(M) & = 0
\end{array}
$$

The first rule initializes the translation, by computing the checks that can be made on the initial knowledge of the strands. The second rule says that each received message is associated with a new label variable $\mathcal{X}_{|M|+1}$ in the agent's knowledge and afterwards we use *ccs* to generate all the checks that the agent can perform on the augmented knowledge. The third rule is for sending the SPS protocol message t. Here we use the relation $M \vdash t^l$ to require that the agent can generate the required term t from the current knowledge M using the concrete sequence of actions l; this is explained in more detail below. The event rule is very similar to sending. The fifth rule translates the construct $\mathsf{fresh}\ X$: we simply pick a new label variable $\mathcal{X}_{|M|+1}$ that will store the fresh value in the translated strand, and bind it in the knowledge to the protocol variable X. The final rule is straightforward.

Example 3.3 *Let us continue on Example 3.2, where we considered an agent with knowledge* $M = [\mathcal{X}_1 \mapsto \mathtt{k}, \mathcal{X}_2 \mapsto \mathtt{X}, \mathcal{X}_3 \mapsto \mathtt{scrypt}(\mathtt{k}, \mathsf{exp}(\mathsf{g}, \mathtt{Y}))]$. *(As explained above, this may result from a strand that initially knows a key in* $\mathcal{X}_1$*, has freshly generated an exponent* $\mathcal{X}_2$*, and has received the message* $\mathcal{X}_3$*.) Suppose that the next step is* $\mathsf{send}(\mathsf{insec}, \mathsf{exp}(\mathsf{exp}(\mathsf{g}, \mathtt{X}), \mathtt{Y}))$ *(in fact, in a more realistic example, it would be a message encrypted with this term as a key). The semantics tells us to determine any label* l *such that* $M \vdash \mathsf{exp}(\mathsf{exp}(\mathsf{g}, \mathtt{X}), \mathtt{Y})^l$*, which is possible for the label* $l = \mathsf{exp}(\mathtt{dscrypt}(\mathcal{X}_1, \mathcal{X}_3), \mathcal{X}_2)$ *as shown in the previous example. Thus, a possible translation is* $\mathsf{send}(\mathsf{insec}, \mathsf{exp}(\mathtt{dscrypt}(\mathcal{X}_1, \mathcal{X}_3), \mathcal{X}_2))$*. Note that we said "*possible*" here, because there are other labels, e.g., any label* l' *such that* $l \approx l'$*.* [2]

More generally, given M and t, there is in general not a unique l such that $M \vdash t^l$. First, consider the case that there is no such l. In this case, the agent has no means (within the deduction relation) to obtain the term t from its current knowledge. We thus say the protocol is *non-executable* and its semantics is

[2] It is common to attach to a Diffie-Hellman exchange also a zero-knowledge proof that the sender knows indeed the exponent behind his half-key. This could easily be modeled by our equational theory with $\mathtt{vexp}(\mathsf{g}, \mathsf{exp}(\mathsf{g}, \mathtt{X})) \approx \top$ for a new operator $\mathtt{vexp}$: then honest agents would only accept exponentiations of the "proper form" as half-keys [CEvdGP87].

undefined. This executability check is an important sanity check on SPS specifications, ensuring that all the steps of the protocol can actually be performed at least when no intruder is interfering and the network does not loose messages. Other formal specification language like Applied π that specify the different roles separately as processes cannot have such an executability check, because unlike SPS, there is no formal relationship between the messages that one role is sending and another is receiving. Thus, if a modeler accidentally specifies messages slightly differently in two processes, they may be unable to communicate and get stuck in their execution; then a flawed protocol may be trivially verified as secure because of the specification mistake. The executability check in SPS drastically reduces the chance of such mistakes.

Second, if there is a label l, then there will typically be infinitely many of them (trivially by performing redundant encryptions and decryptions). Our semantics does not prescribe which of the labels has to be taken (and the implementation below will take in some sense the simplest one). A key insight is that this does not make the semantics ambiguous: if $M \vdash t^{l_1}$ and $M \vdash t^{l_2}$ then $ccs(M) \models l_1 \doteq l_2$. Thus, since we always perform the checks on the knowledge after each received message, we know that the choice of labels does not make a difference.

As an example, observe that the operational strand we have given in Figure 3.1(right) for our example protocol is correct according to this semantics (when resolving the $\mathcal{X} := t$ macros): all outgoing messages have an appropriate label (for which $M \vdash t^l$ holds), and all checks $s \doteq t$ do indeed logically follow from $ccs(M)$ for the respective M. In fact, we claim that the checks are logically equivalent to $ccs(M)$, i.e., all other checks are redundant; it is part of the results of the next section to prove that and derive the given checks automatically.

We emphasize the succinctness of the definitions: Definitions 3.1–3.5 together fit on a page and yet we define the semantics for an arbitrary set of cryptographic operators and algebraic properties. We believe that this is the best argument that the semantics of Alice-and-Bob notation should be defined this way—deriving from simple, general, and uniform principles. However, this simple semantics cannot be directly used as a translator from Alice-and-Bob notation to formal models or implementations as it entails an infinite representation and several of the underlying algebraic problems are in fact not recursive in general.

Theorem 1 *The problem to compute a finite representation of $[\![S]\!]_H$ for a strand S, if it exists, is not recursive.*

Proof. First, pick any undecidable $\approx$, then follows immediately that the set $A_\approx = \{(s,t) \mid s \approx t\}$ is also undecidable. Now, assume that $B_{ccs} =$

$\{(M, \mathcal{X}_1, \mathcal{X}_2) \mid ccs(M) \models \mathcal{X}_1 \doteq \mathcal{X}_2\}$ is decidable. Let $h(s,t) = ([\mathcal{X}_1 \mapsto s, \mathcal{X}_2 \mapsto t], \mathcal{X}_1, \mathcal{X}_2)$ that is obviously computable. Moreover, h reduces $A_{\approx}$ to B_{ccs} since: if $(s,t) \in A_{\approx}$, then from the definition of $ccs(M)$ follows that $ccs(M) \models s \doteq t$ holds, therefore $h(s,t) \in B_{ccs}$, and if $(s,t) \notin A_{\approx}$, then from the definition of $ccs(M)$ follows that $ccs(M) \models s \doteq t$ does not hold, and therefore $h(s,t) \notin B_{ccs}$. We reach a contradiction, and therefore B_{ccs} is undecidable. It follows that for a given knowledge M, the problem to compute a finite conjunction ϕ, such that $\phi \equiv ccs(M)$, if one exists, is not recursive.

Similarly, we can prove that $\vdash$ is in general an undecidable relation. Let $B_{\vdash} = \{(M,t,l) \mid M \vdash t^l\}$ and assume that it is decidable. Let $g(s,t) = ([\mathcal{X}_1 \mapsto s], t, \mathcal{X}_1)$ that is also obviously computable and reduces $A_{\approx}$ to $B_{\vdash}$ as follows: if $(s,t) \in A_{\approx}$, then from the definition of $\vdash$ follows that $[\mathcal{X}_1 \mapsto s] \vdash t^{\mathcal{X}_1}$ holds, therefore $g(s,t) \in B_{\vdash}$, and if $(s,t) \notin A_{\approx}$, then from the definition of $\vdash$ follows that $[\mathcal{X}_1 \mapsto s] \vdash t^{\mathcal{X}_1}$ does not hold, therefore $g(s,t) \notin B_{\vdash}$. Again, we reach a contradiction, and therefore $B_{\vdash}$ is undecidable. It follows that for a given knowledge M and a term t, the problem to compute a label l, such that $M \vdash t^l$, if one exists, is not recursive. □

3.4 Low-level Semantics

In this section, we define the procedures for message composition and decomposition (*compose* and *analyze* respectively), but we first need some necessary definitions. First, we need to partition the public operations into constructors and destructors.

Definition 3.6 Let $\Sigma_d = \{\texttt{dscrypt}, \texttt{vscrypt}, \texttt{dcrypt}, \texttt{vcrypt}, \texttt{open}, \texttt{vsign}, \texttt{get}_., \texttt{verify}_.\}$ be the *destructors*, where, abusing notation, we include $\texttt{get}_.$ and $\texttt{verify}_.$ for all formats. All other public operators $\Sigma_c = \Sigma_p \setminus \Sigma_d$ are called *constructors*. Let finally $\Sigma_A = \Sigma \setminus \Sigma_d$ contain all symbols except destructors. We require that the terms in the SPS specification are from $\mathcal{T}_{\Sigma_A}(V)$ (except label variables in the initial knowledge).

Let us also denote by $\approx_F$ the least congruence relation that satisfies properties (9)-(11) shown in Table 3.1 (which address modular exponentiation and multiplication). Since we have here no destructors for `exp` and `mult`, $\approx_F$ is a *finite* theory; i.e., for any term t, the equivalence class of t under $\approx_F$ is finite (moreover, unification is *finitary*, i.e., we can find finitely many most general unifiers for every pair of terms). We also define $\vdash_C$ as a restriction of $\vdash$ (Definition 3.3) where $\approx$ is replaced with $\approx_F$ and restricting Σ_p to Σ_c. Thus, $\vdash_C$

is the "compositional" part of the $\vdash$ relation that allows only composing terms and application of $\approx_F$ (which never "decomposes" terms).

3.4.1 Message Composition

We now define the compositional part of message deduction, i.e., computing $\vdash_C$, realized by the function $\mathit{compose}_M(t)$ that computes all labels for generating the term t from knowledge M using only $\vdash_C$.

Definition 3.7 Let M be a knowledge and $t \in \mathcal{T}_{\Sigma_A}(V)$.

$$\begin{aligned}\mathit{compose}_M(t) = \{\mathcal{X}_i \mid \exists t'.\ [\mathcal{X}_i \mapsto t'] \in M \wedge t \approx_F t'\} \cup \\ \{f(l_1,\ldots,l_n) \mid \exists t_1,\ldots,t_n.\ t \approx_F f(t_1,\ldots,t_n) \wedge f \in \Sigma_c \wedge \\ l_1 \in \mathit{compose}_M(t_1) \wedge \ldots \wedge l_n \in \mathit{compose}_M(t_n)\}\,.\end{aligned}$$

The first part of $\mathit{compose}_M$ checks whether the term t is directly contained in the knowledge modulo $\approx_F$, and returns corresponding label variables if so. The second part computes all ways to recursively compose t from its direct subterms (modulo $\approx_F$). For instance, for $M = [\mathcal{X}_1 \mapsto c, \mathcal{X}_2 \mapsto \mathsf{hash}(c)]$ we have $\mathit{compose}_M(\mathsf{hash}(c)) = \{\mathcal{X}_2, \mathsf{hash}(\mathcal{X}_1)\}$, and for $M = [\mathcal{X}_1 \mapsto a \cdot b, \mathcal{X}_2 \mapsto c, \mathcal{X}_3 \mapsto a \cdot c, \mathcal{X}_4 \mapsto b]$ (writing $a \cdot b$ for $\mathtt{mult}(a,b)$), we have $\mathit{compose}_M(a \cdot b \cdot c) = \{\mathcal{X}_1 \cdot \mathcal{X}_2, \mathcal{X}_3 \cdot \mathcal{X}_4\}$.

The $\mathit{compose}_M$ function does not involve any decomposition steps or generate checks—for this we define an analysis procedure in the next subsection. The interface between the two procedures is the notion of an *analyzed* knowledge (in which every possible analysis step has already been done). We define this notion succinctly by requiring that every term that can be derived from M using $\vdash$ can also be derived using $\vdash_C$, i.e., analysis steps do not yield any further messages:

Definition 3.8 We say a knowledge M is *analyzed* iff

$$\{t \in \mathcal{T}_{\Sigma_A}(V) \mid \mathit{exists}\ l.\ M \vdash t^l\} = \{t \in \mathcal{T}_{\Sigma_A}(V) \mid \mathit{exists}\ l.\ M \vdash_C t^l\}.$$

For an analyzed knowledge M, $\mathit{compose}_M$ is in fact correct:

Theorem 2 *The $\mathit{compose}_M$ function terminates and is sound in the sense that $l \in \mathit{compose}_M(t)$ implies $M \vdash t^l$. Moreover, if M is analyzed and neither M nor t contain symbols from Σ_d, then $\mathit{compose}_M$ is also complete in the sense that $M \vdash t^l$ implies $l' \in \mathit{compose}_M(t)$ for some label l' with $\mathit{ccs}(M) \models l \doteq l'$.*

PROOF. For *Termination*, consider the tree of recursive calls that $compose_M(t)$ can invoke. The tree is finitely branching since $\approx_F$ is a finite theory (every term has a finite equivalence class). Suppose the tree has infinite depth, and let $t_1, t_2, t_3, \ldots$ be the terms in the recursive calls. Then there are terms $t'_1, t'_2, t'_3, \ldots$ such that $t_i \approx_F t'_i \sqsupset t_{i+1}$ for all $i \geq 1$. Then there are contexts $C_1[\cdot], C_2[\cdot], \ldots$ and $C_i[x] \neq x$ such that $t_1 \approx_F C_1[t_2] \approx_F C_1[C_2[t_3]] \approx_F \ldots$ and thus t_1 has an infinite equivalence class modulo $\approx_F$, which is absurd, so the tree also has finite depth.

Soundness is immediate as we assume that functions are treated as strict, i.e., when recursively building a composition and the result for any subterm is $\emptyset$ then the whole expression is $\emptyset$. For each step in $compose_M$ it is straightforward that it is covered by rules in the inductive definition of $\vdash$.

For *Completeness*, consider $M \vdash t^l$, where M is analyzed and M and t do not contain any symbols from Σ_d. Since M is analyzed, we also have $M \vdash_C t^{l'}$ for some l', and thus $ccs(M) \models l \doteq l'$. Due to $\vdash_C$, l' cannot contain any symbol from Σ_d either (while l can). Consider now the proof tree for $M \vdash_C t^{l'}$: leaf nodes are axioms and inner nodes are either composition steps with $f \in \Sigma_c$ or algebraic equivalences modulo $\approx_F$. It is straightforward to map them into corresponding steps of $compose_M(t)$ to yield label l'. □

3.4.2 Message Decomposition and Checks

To compute an analyzed knowledge and the checks that one can perform on it, we define the procedure *analyze* that takes as input a pair (M, φ) of a knowledge and a (finite) conjunction of equations and yields a *saturated* extension $(M \cup M', \varphi \wedge \varphi')$ of (M, φ). The notion of *saturated* means that $M \cup M'$ is analyzed and that $\varphi \wedge \varphi'$ is equivalent to $ccs(M \cup M')$. Note that this algorithm works incrementally, so when augmenting M with a received message in the generation of operational strands, we do not need to start the analysis from scratch. Also, we assume that we never add redundant checks, i.e., the ones that are already entailed by previous checks.

1. Term of the form	2. Condition	3. Derive	4. Check	5. Recipe
$\mathcal{X}_i \mapsto \mathtt{scrypt}(k, m)$	$l \in \mathit{compose}_M(k)$	$\mathcal{X}_{i'} \mapsto m$	$\mathsf{vscrypt}(l, \mathcal{X}_i) \doteq \top$	$\mathcal{X}_{i'} := \mathtt{dscrypt}(l, \mathcal{X}_i)$
$\mathcal{X}_i \mapsto \mathtt{crypt}(k, m)$	$l \in \mathit{compose}_M(\mathtt{inv}(k))$	$\mathcal{X}_{i'} \mapsto m$	$\mathsf{vcrypt}(l, \mathcal{X}_i) \doteq \top$	$\mathcal{X}_{i'} := \mathtt{dcrypt}(l, \mathcal{X}_i)$
$\mathcal{X}_i \mapsto \mathtt{sign}(\mathtt{inv}(k), m)$	$l \in \mathit{compose}_M(k)$	$\mathcal{X}_{i'} \mapsto m$	$\mathtt{vsign}(l, \mathcal{X}_i) \doteq \top$	$\mathcal{X}_{i'} := \mathtt{open}(l, \mathcal{X}_i)$
$\mathcal{X}_i \mapsto \mathtt{f}(t_1, \ldots, t_n)$ for some $\mathtt{f} \in \Sigma_{\mathtt{f}}$	(true)	$\mathcal{X}_{i'_1} \mapsto t_1 \ \ldots$ $\mathcal{X}_{i'_n} \mapsto t_n$	$\mathsf{verify}_{\mathtt{f}}(\mathcal{X}_i) \doteq \top$	$\mathcal{X}_{i'_1} := \mathsf{get}_{1,\mathtt{f}}(\mathcal{X}_i) \ \ldots$ $\mathcal{X}_{i'_n} := \mathsf{get}_{n,\mathtt{f}}(\mathcal{X}_i)$

let in the following $\frac{t_d}{s_d} = \mathit{shorten}(\frac{t_1 \cdot \ldots \cdot t_n}{s_1 \cdot \ldots \cdot s_m})$ and $i \neq j$

1. Term of the form $(i \neq j)$	2. Condition	3. Check
$\mathcal{X}_i \mapsto t$	$\{l_1, \ldots, l_n\} = \mathit{compose}_M(t)$	$l_1 \doteq l_2 \doteq \ldots \doteq l_n$
$\mathcal{X}_i \mapsto \mathtt{inv}(k)$	$l \in \mathit{compose}_M(k)$	$\mathsf{vcrypt}(\mathcal{X}_i, \mathtt{crypt}(l, \top)) \doteq \top$
$\mathcal{X}_i \mapsto t_1 \cdot \ldots \cdot t_n$ $\mathcal{X}_j \mapsto s_1 \cdot \ldots \cdot s_m$	$l_{t_d} \in \mathit{compose}_M(t_d)$ $l_{s_d} \in \mathit{compose}_M(s_d)$	$l_{t_d} \cdot \mathcal{X}_j \doteq l_{s_d} \cdot \mathcal{X}_i$
$\mathcal{X}_i \mapsto \mathsf{exp}(A, t_1 \cdot \ldots \cdot t_n)$ $\mathcal{X}_j \mapsto \mathsf{exp}(A, s_1 \cdot \ldots \cdot s_m)$	$l_{t_d} \in \mathit{compose}_M(t_d)$ $l_{s_d} \in \mathit{compose}_M(s_d)$	$\mathsf{exp}(\mathcal{X}_j, l_{t_d}) \doteq \mathsf{exp}(\mathcal{X}_i, l_{s_d})$
$\mathcal{X}_i \mapsto \mathsf{exp}(A, t_1 \cdot \ldots \cdot t_n)$ $\mathcal{X}_j \mapsto s_1 \cdot \ldots \cdot s_m$	$l_{t_d} \in \mathit{compose}_M(t_d)$ $l_A \in \mathit{compose}_M(A)$ $l_{s_d} \in \mathit{compose}_M(s_d)$	$\mathsf{exp}(l_A, \mathcal{X}_j \cdot l_{t_d}) \doteq \mathsf{exp}(\mathcal{X}_i, l_{s_d})$

Table 3.2: Tabular overview of $\mathit{analyze}(M, \varphi)$

Table 3.2 summarizes the procedure $analyze(M, \varphi)$. The table is divided into two parts: the upper part represents the first phase of the algorithm, saturating M with derivable subterms, whereas the lower part represents the second phase saturating φ with additional equations.

Phase 1. Here we check for every entry in M whether it can be analyzed, i.e., if it has one of the forms of column 1 (the head symbol being `scrypt`, `crypt`, `sign`, or a format) and that has not yet been marked as analyzed (initially no term is). We then check according to column 2 whether the necessary decryption key can be derived. For this, we use the $compose_M$ procedure yielding a label l if the key is available; if there are several labels, we simply pick one. We then mark the entry $\mathcal{X}_i \mapsto \ldots$ as analyzed, choose a new label variable $\mathcal{X}_{i'}$ and add the analyzed message to the knowledge M according to column 3. Further, we add the condition of column 4 and the recipe of column 5 to ϕ. (We treat the recipe here like an equation for simplicity.) We repeat this until no more analysis step can be performed. (Note: whenever new terms are added to M, encrypted messages that have not been marked as analyzed need to be checked again.)

Phase 2. We now consider every entry of M once and check for all alternative ways to generate it according to the first row in the lower part of the table. If we find more than one such label, we add the respective checks to ϕ. The second row is to check if a private key fits to its corresponding public key if it is known.[3]

Next, we have rules for products and exponents (last three rows of the lower table). Here we consider any pair of entries in M where the head symbol is `exp` or `mult` (according to the form of column 1), again writing $\cdot$ for multiplication. Here, we require a match such that none of the s_i and t_i is itself a product. We then consider the fraction $(t_1 \cdot \ldots \cdot t_n)/(s_1 \cdot \ldots \cdot s_m)$ and *shorten* it, i.e., removing common factors in enumerator and denominator. Let t_d/s_d be the remaining products after shortening. If all the t_i or all the s_i are shortened away (i.e., $t_d = 1$ or $ts_d = 1$) we do not apply this rule (as it is already covered by the first row, saving us from introducing 1 into the algebraic theory). We now try to compose the products t_d and s_d according to column 2. If there is at least one label for each (if there are several, again we pick one), then we add to ϕ the condition of column 3.

Example 3.4 *We compute* $analyze(M, \top)$ *for the knowledge* $M = [\mathcal{X}_1 \mapsto y,$

[3] This check is actually quite academic, as the agent has either generated the key pair itself (and thus knows by construction that they form a key pair) or it has received it from a key server, e.g., in identity-based encryption (but then needs to trust that server anyway). However, without this check the correctness theorem and its proof would require a more complicated formulation.

$\mathcal{X}_2 \mapsto \mathtt{scrypt}(\mathsf{exp}(\mathsf{exp}(g,y),x),n), \mathcal{X}_3 \mapsto \mathtt{scrypt}(k,\mathsf{exp}(g,x)), \mathcal{X}_4 \mapsto k, \mathcal{X}_5 \mapsto \mathsf{hash}(n)]$.

For phase 1, entries $\mathcal{X}_1$, $\mathcal{X}_4$, and $\mathcal{X}_5$ do not match any entry in the first column (they cannot possibly be decrypted). For $\mathcal{X}_2$, we have $compose_M(\mathsf{exp}(\mathsf{exp}(g,y),x)) = \emptyset$, i.e., the decryption key is not (yet) available. However, we can decrypt $\mathcal{X}_3$ since $compose_M(k) = \{\mathcal{X}_4\}$. We thus add $\mathcal{X}_6 \mapsto \mathsf{exp}(g,x)$ to the knowledge, and to ϕ the check $\mathsf{vscrypt}(\mathcal{X}_4,\mathcal{X}_3) \doteq \top$ and the recipe $\mathcal{X}_6 := \mathtt{dscrypt}(\mathcal{X}_4,X_3)$. We mark $\mathcal{X}_3$ as analyzed, and check again the unanalyzed $\mathcal{X}_2$. This time (for the updated M) we have $compose_M(\mathsf{exp}(\mathsf{exp}(g,y),x)) = \{\mathsf{exp}(\mathcal{X}_6,\mathcal{X}_1)\}$, and thus add $\mathcal{X}_7 \mapsto n$ to the knowledge, and to ϕ the check $\mathsf{vscrypt}(\mathsf{exp}(\mathcal{X}_6,\mathcal{X}_1),\mathcal{X}_2) \doteq \top$ and recipe $\mathcal{X}_7 := \mathtt{dscrypt}(\mathsf{exp}(\mathcal{X}_6,\mathcal{X}_1),\mathcal{X}_2)$. Since neither $\mathcal{X}_6$ and $\mathcal{X}_7$ can be further analyzed, phase 1 is finished. For phase 2, we can of course re-construct the encryptions, e.g., $\mathtt{scrypt}(\mathcal{X}_4,\mathcal{X}_6) \doteq \mathcal{X}_3$ but that is already implied by the equation $\mathcal{X}_6 := \mathtt{dscrypt}(\mathcal{X}_4,\mathcal{X}_3)$ and we do not add redundant checks. The only new check is for $\mathcal{X}_5$, since $compose_M(\mathsf{hash}(n)) = \{\mathcal{X}_5, \mathsf{hash}(\mathcal{X}_7)\}$ yields $\mathcal{X}_5 \doteq \mathsf{hash}(\mathcal{X}_7)$.

As another example for equational reasoning, $analyze([\mathcal{X}_1 \mapsto a \cdot b \cdot c, \mathcal{X}_2 \mapsto a \cdot c \cdot d, \mathcal{X}_3 \mapsto b, \mathcal{X}_4 \mapsto d], \top)$ yields the check $\mathcal{X}_1 \cdot \mathcal{X}_4 \doteq \mathcal{X}_2 \cdot \mathcal{X}_3$. □

Theorem 3 *For a knowledge M with no symbols in Σ_d and a finite conjunction ϕ of equations, $analyze(M,\phi)$ terminates with a result (M',ϕ') where $M' = M \cup [\mathcal{X}_{|M|+1} \mapsto t_1, \cdots, \mathcal{X}_{|M|+n} \mapsto t_n]$, $\phi' = \phi \wedge \mathcal{X}_{|M|+1} := l_1 \wedge \cdots \wedge \mathcal{X}_{|M|+n} := l_n \wedge \psi$ and $M \vdash t_1^{l_1}, \cdots, M \vdash t_n^{l_n}$ such that $\{t \mid exists\ l.M \vdash t^l\} = \{t \mid exists\ l.M' \vdash t^l\}$ (soundness), $analyzed(M')$ (completeness), and $ccs(M) \equiv \phi \wedge \psi$ (correctness of checks).*

Proof. *Termination:* The newly added terms of M' are always subterms of some term in M, so the M' component must eventually reach a fixed point. Adding new equations to ϕ' is bounded by pairs of entries of M' and the finiteness of $compose_M$.

Soundness: This is immediate, as in all cases of *analyze* we merely add derivable messages to the knowledge M, i.e., we can check that we add only new messages that are really derivable from M.

Completeness: We first show that M' is analyzed, i.e., we have to show that for any term $t \in \mathcal{T}_{\Sigma_A}(V)$ with $M' \vdash t^l$, we also have $M' \vdash_C t^{l'}$ for some l' (i.e., using only constructors of Σ_c and equivalence in $\approx_F$). For this, we consider the proof tree for $M' \vdash t$. Intermediate nodes may well contain destructors, but we can exclude so-called *garbage terms*, namely terms that are

not $\approx$-equivalent to any term in $\mathcal{T}_{\Sigma_A}(V)$. For instance, $\mathtt{dscrypt}(c,c)$ is garbage (while $\mathtt{dscrypt}(k,\mathtt{scrypt}(k,m)) \approx m$ is not for $m \in \mathcal{T}_{\Sigma_A}(V)$). Suppose the proof contains a node with a garbage term s, then there must be a construction in the proof to remove s (since the final term must be in $\mathcal{T}_{\Sigma_A}(V)$), for instance constructing $\mathtt{dscrypt}(s,\mathtt{scrypt}(s,m)) \approx m$ eliminates garbage s, but in all such cases, all occurrences of s must have been composed, so there exists a simpler proof without garbage.

We thus first show the following: for any $M' \vdash t^l$ where t is not garbage we have $M' \vdash_C s^k$ for some $s \approx t$ and some label k. This is shown by induction over the proof tree of $M' \vdash t^l$. For *Ax* and *Eq* the proof is immediate as well as for *Cmp* with $f \in \Sigma_c$. For $f \in \Sigma_d$, consider the term t_0 being analyzed. By induction $M' \vdash_C s_0^{k_0}$ for some $s_0 \approx t_0$, so is (modulo $\approx_F$) either composed or an axiom. If it is composed, then the intruder decomposes a term he has composed himself and this proof can be simplified. If it is an axiom, then the intruder applies decomposition to a term in his knowledge, and *analyze* has already added the resulting term t (modulo $\approx$) to M'.

Note we have only proved that for $M' \vdash t^l$ (where t is not garbage) there is $M' \vdash_C s^k$ for some $s \approx t$. We have to show that $M' \vdash_C t^{l'}$ for some l', but only for $t \in \mathcal{T}_{\Sigma_A}(V)$, i.e., without destructors. We claim that in this case we have $s \approx_F t$ (and thus follows $M' \vdash_C t^k$ as $\vdash_C$ is closed under $\approx_F$). This claim follows from the fact that our destructor equations (1)–(8) can be read as rewrite rules (from left to right) that are convergent modulo $\approx_F$, and thus terms that do not contain constructors are in normal form modulo $\approx_F$. The idea for proving this convergence is that the rewriting rules have disjoint symbols from the equations in $\approx_F$ (so they cannot conflict) and we can prove convergence for the rewrite rules using the critical pair method, see e.g. [BN98].

Correctness of Checks: Now for $ccs(M) \equiv \phi \wedge \psi$. For brevity let $\psi' = \phi \wedge \psi$. The soundness ($ccs(M) \implies \psi'$) is obvious by checking that each step in *analyze* adds only sound equations. The completeness we prove again indirectly, i.e., suppose we have a term t and two derivation proofs $M \vdash t^l$ and $M \vdash t^{l'}$ such that $l \doteq l'$ is not implied by ψ'. Suppose in either of the derivation trees for l and l' appears a composition step with a destructor. Suppose the message being decomposed is $t_1^{l_1}$ and the result of decomposing is $t_0^{l_0}$. Again assume that there are no decompositions in the subtrees. One possible case is the analysis of `inv` which is covered by the sixth case in *analyze* (Table 3.2). In all other cases, $analyze(M,\phi)$ must have added t_0 under some new label X_i to M' and ψ' must entail $X_i \doteq l_0$ (and a constraint about verifiability of l_1). Let us thus replace the derivation $t_0^{l_0}$ with $t_0^{X_i}$: this changes a subterm in labels l and l', but for these changed labels still $l \doteq l'$ does not follow from ψ'. In this way we can step by step eliminate all analysis steps and thus have two trees without analysis for t^l and $t^{l'}$ such that $l \doteq l'$ is not implied by ψ'.

Now we consider the case that either of the two trees (say for l') is an application of only axiom and equality steps, thus $l' = X_i$ for some variable X_i. Then M' contains $[X_i \mapsto t]$ for a term that can be composed in a different way using only constructors and $\approx_F$, i.e., $l \in \mathit{compose}_M(t)$ and thus ψ' must contain $l \doteq l'$, contradicting the assumption. Otherwise it must be two trees consisting of composition steps. We can exclude composition with any operator but `exp` or `mult` since otherwise we can simply go to one of the subterms. If it is `exp` or `mult`, then it has the form of adding factors to initially known `exp` or `mult` terms. Again we can exclude adding the same factor in both trees (since otherwise we can reduce again to a simpler case). The remaining case is however covered by our check rules for `exp` and `mult`, again showing that $l \approx l'$ must be entailed by ψ'.

3.4.3 Implementing the Semantics

Now we put it all together to define a more low-level, procedural semantics that computes the semantics for the example theory in Table 3.1 based on *compose* and *analyze* procedures. We require that the SPS specification (and thus the plain strands) does not contain any destructors or verifiers. We prove later that this low-level semantics is actually computable and prove that it correctly implements the high-level semantics.

We define the following computable low-level semantics that translates from plain strands to operational strands and mirrors the structure of the high-level semantics as follows.

Definition 3.9 Given a plain strand S with no destructors or verifiers, $[\![S]\!]_L$ translates S to an operational strand as follows with the initial rule $[\![\cdot]\!]_{L_0}$

$$
\begin{aligned}
[\![M : \mathit{steps}]\!]_{L_0}(\emptyset, \top) &= M : \varphi.[\![\mathit{steps}]\!]_L(M', \varphi) \\
&\quad \text{where } (M', \varphi) = \mathit{analyze}(M, \top) \\
[\![\mathsf{receive}(ch, t).\mathit{rest}]\!]_L(M, \varphi) &= \mathsf{receive}(ch, \mathcal{X}_{|M|+1}).\varphi'.[\![\mathit{rest}]\!]_L(M', (\varphi \wedge \varphi')) \\
&\quad \text{where } (M', \varphi \wedge \varphi') = \mathit{analyze}(M \cup [\mathcal{X}_{|M|+1} \mapsto t], \varphi) \\
[\![\mathsf{send}(ch, t).\mathit{rest}]\!]_L(M, \varphi) &= \mathsf{send}(ch, l).[\![\mathit{rest}]\!]_L(M, \varphi) \\
&\quad \text{where } l \in \mathit{compose}_M(t) \\
[\![\mathsf{event}(t).\mathit{rest}]\!]_L(M, \varphi) &= \mathsf{event}(l).[\![\mathit{rest}]\!]_L(M, \varphi) \\
&\quad \text{where } l \in \mathit{compose}_M(t) \\
[\![\mathsf{fresh}\ X.\mathit{rest}]\!]_L(M, \varphi) &= \mathsf{fresh}\ \mathcal{X}_{|M|+1}.[\![\mathit{rest}]\!]_L(M \cup \{\mathcal{X}_{|M|+1} \mapsto X\}, \varphi) \\
[\![0]\!]_L(M, \varphi) &= 0
\end{aligned}
$$

Theorem 4 *For our example theory in Table 3.1, for every strand S in which no destructors or verifiers occur, $[\![S]\!]_L$ is recursive and has a finite representation.*

Proof. First, by Theorem 2, *compose* is recursive and produces a finite set of labels. Second, by Theorem 3, *analyze* is recursive and produces a finite conjunction of checks and a finite knowledge. Finally, given a strand S that is finite (being defined by context-free grammar), one can easily see from the rules of $[\![\cdot]\!]_L$ and from the previous two points that $[\![\cdot]\!]_L$ is recursive and it has a finite representation. □

Now we proceed by showing that the two levels of our semantics ($[\![\cdot]\!]_H$ and $[\![\cdot]\!]_L$) coincide, i.e., given the same plain strand as input, they produce in some sense equivalent operational strands.

3.4.4 Equivalence of Strands

The missing point now is the connection between the two semanticses ($[\![\cdot]\!]_H$ and $[\![\cdot]\!]_L$), i.e., given the same plain strand, whether they produce equivalent operational strands. We thus need to define a notion of *equivalence* between strands. Intuitively, two strands are equivalent if they have the same initial knowledge, corresponding send and receive steps, equivalent checks and events. Based on this notion of equivalence, we now discuss the rules of the two semantics to show that they produce equivalent operational strands.

- The initial case:

$$[\![M : strand]\!]_{H_0} \quad = M : ccs(M).[\![strand]\!]_H(M)$$

compared with the corresponding rule:

$$[\![M : strand]\!]_{L_0}(\emptyset, \top) \quad = M : \varphi.[\![strand]\!]_L(M', \varphi), \\ \text{where } (M', \varphi) = analyze(M, \top)$$

The first difference between the results of the two rules is the checks, i.e., in the first one we have $ccs(M)$ and in the second one we have φ that is the conjunction of checks that *analyze* procedure produces. By Theorem 3 we have that they are equivalent, i.e., $ccs(M) \equiv \varphi$ in our case. The second difference is the knowledge carried out for the next steps, i.e., in the first rule we have M while in the second rule we have M' that is an

analyzed version of M. Recall that a knowledge and its analyzed version are equivalent in a sense that one can derive the same terms from both. The main difference between the two versions of a knowledge (the original and the analyzed) is that the analyzed version has no further analysis steps and this is needed for the termination of *compose* (cf. Definitions 3.7 and 3.8).

- send case:

$$[\![\mathsf{send}(ch,t).rest]\!]_H(M) \quad = \mathsf{send}(ch,l).[\![rest]\!]_H(M), \\ \text{where } M \vdash t^l \text{ for some label } l$$

compared with the corresponding rule:

$$[\![\mathsf{send}(ch,t).rest]\!]_L(M,\varphi) \quad = \mathsf{send}(ch,l).[\![rest]\!]_L(M,\varphi) \\ \text{where } l \in compose_M(t)$$

The only difference between the two rules is the way the recipe l is derived, i.e., in the first rule we have $M \vdash t^l$ and in the second rule we have $l \in compose_M(t)$. By Theorem 2 (soundness and completeness of *compose*) and by Theorem 3 we have that we either have the same label for t, or if we have different labels for t (say l and l') then a check must be added to reflect that $(l \doteq l')$ cf. Definition 3.4.

- event case: similar to the send case.
- receive case:

$$[\![\mathsf{receive}(ch,t).rest]\!]_H(M) \quad = \mathsf{receive}(ch,\mathcal{X}_{|M|+1}).ccs(M \cup [\mathcal{X}_{|M|+1} \mapsto t]). \\ [\![rest]\!]_H(M \cup [\mathcal{X}_{|M|+1} \mapsto t])$$

compared with the corresponding rule:

$$[\![\mathsf{receive}(ch,t).rest]\!]_L(M,\varphi) \quad = \mathsf{receive}(ch,\mathcal{X}_{|M|+1}).\varphi'.[\![rest]\!]_L(M',(\varphi \wedge \varphi')), \\ \text{where } (M',\varphi \wedge \varphi') = analyze(M \cup [\mathcal{X}_{|M|+1} \mapsto t],\varphi)$$

The difference between the two rules is the check and knowledge parts. For the checks part, note that by Theorem 3, we have $ccs(M) \equiv \varphi$ and $ccs(M \cup [\mathcal{X}_{|M|+1} \mapsto t]) \equiv \varphi \wedge \varphi'$. For the knowledge part, note that M' is an analyzed version of $M \cup [\mathcal{X}_{|M|+1} \mapsto t]$ as the later rule indicates. Note that the labels could be different; because an analyzed knowledge has in general more terms than the original version of it. Therefore, the result of these two rules may not be identical as they may be receiving the term t with different labels. However, a proper α-renaming can resolve that and

make the two resulting strands identical; except that the knowledge of one of them is the analyzed version of the other one, but in principle they are still equivalent, i.e., one can derive the same terms from both.

- fresh case:

$$[\![\mathsf{fresh}\ X.rest]\!]_H(M) \quad = \mathsf{fresh}\ \mathcal{X}_{|M|+1}.[\![rest]\!]_H(M \cup \{\mathcal{X}_{|M|+1} \mapsto X\})$$

with the corresponding rule:

$$[\![\mathsf{fresh}\ X.rest]\!]_L(M,\varphi) \quad = \mathsf{fresh}\ \mathcal{X}_{|M|+1}.[\![rest]\!]_L(M \cup \{\mathcal{X}_{|M|+1} \mapsto X\},\varphi)$$

Again, the only difference that may occur between the two is the label of the fresh value X, but as we discussed in the previous case, a proper α-renaming can resolve it with no semantical effect.

By this, we can conclude that for our example theory in Table 3.1, $[\![\cdot]\!]_L$ is an implementation of $[\![\cdot]\!]_H$.

Theorem 5 *For our example theory in Table 3.1, for every strand S in which no destructors or verifiers occur, $[\![S]\!]_H$ can be finitely represented and it is recursive.*

Proof. We have just shown that for a given strand S with no destructors nor verifiers $[\![S]\!]_H$ is equivalent to $[\![S]\!]_L$. We also proved in Theorem 4 that $[\![\cdot]\!]_L$ is recursive and has a finite representation. □

3.5 Operational Strands Semantics

We conclude the semantics chapter by giving the semantics of operational strands. Similar to [CM05], we define the semantics of operational strands as an infinite-state transition system, where a state $(\mathcal{S};\mathcal{M};E)$ consists of (1) a set $\mathcal{S}$ of closed strands, (i.e., every variable occurs first in a receive message, in a macro, or in a creation of a fresh value), (2) a set $\mathcal{M}$ of messages (the intruder knowledge), and (3) a set E of events that have occurred. For instance, if $\mathcal{S}$ contains the strand $\mathsf{send}(\mathsf{insec},t).rest$, where insec represents an insecure channel, then we can make the transition to a successor state where t is added to $\mathcal{M}$ and the send step is removed from the given strand. This transition system is defined by an initial state and a transition relation as follows.

The initial state Recall that in an SPS specification, only variables of type agent may be used in a knowledge declaration; therefore the co-domain of the knowledge M of each operational strand of the protocol will only contain such agent-typed variables. The first step in defining the semantics is to consider all possible instantiations of these agent variables with concrete agent names; and create infinitely many copies of these operational strands to model an unbounded number of sessions between any agents.

Let therefore $\mathcal{S} = \{s_1, \cdots, s_k\}$ be the set of operational strands of a protocol, one for each role of the protocol. Let us further denote by R_i the name of the role (i.e., a constant or variable of type agent) that is described by the operational strand s_i, M_i be the knowledge of s_i and $steps_i$ be the steps of s_i, i.e., $s_i = M_i : steps_i$. Let Ag be a countably infinite set of constants of type `Agent`, including i denoting the intruder, and let V_A be the set of all variables that occur in the M_i (and are thus of type `Agent` in every SPS translation). Let *Subs* be the set of substitutions from V_A to Ag. Thus *Subs* represents all possible instantiations of the roles of the protocol with concrete agent names. If the SPS knowledge declarations contain some inequalities, such as $\mathtt{A} \neq \mathsf{i}$ or $\mathtt{A} \neq \mathtt{B}$, then this set *Subs* is accordingly restricted.

Even though a knowledge itself is a substitution (cf. Definition 3.2), we now define what it means to apply a substitution (from *Subs*) to it. Let $\sigma \in Subs$ and $M = [\mathcal{X}_1 \mapsto t_1, \cdots, \mathcal{X}_l \mapsto t_l]$ be a knowledge. Then, we define $\sigma(M) = [\mathcal{X}_1 \mapsto \sigma(t_1), \cdots, \mathcal{X}_l \mapsto \sigma(t_l)]$. The initial state of the transition system is $(\mathcal{S}_0; \mathcal{M}_0; \emptyset)$ where:

$$\begin{aligned} \mathcal{S}_0 &= \bigcup_{i=1}^{k} \{\sigma(M_i)(steps_i.\mathsf{finished}(n)) \mid \sigma \in Subs, \sigma(R_i) \neq \mathsf{i}, n \in \mathbb{N}\} \\ \mathcal{M}_0 &= \bigcup_{i=1}^{k} \{\sigma(ul(M_i)) \mid \sigma \in Subs, \sigma(R_i) = \mathsf{i}\} \cup Ag \end{aligned}$$

where *ul* is a function that maps from a knowledge to a set of terms by discarding the labels, e.g., $ul([\mathcal{X}_1 \mapsto t_1, \cdots, \mathcal{X}_n \mapsto t_n]) = \{t_1, \cdots, t_n\}$.

Here we use a new event $\mathsf{finished}(n)$ (for each $n \in \mathbb{N}$) to create a countable number of unique operational strands for each instance $\sigma \in Subs$. Note that we apply the instantiation σ first to the knowledge of the role, and the so instantiated knowledge to the entire operational strand. For instance, for the operational strand

$$[\mathcal{X}_1 \mapsto \mathtt{A}, \mathcal{X}_2 \mapsto B, \mathcal{X}_3 \mapsto \mathtt{shk}(\mathtt{A}, \mathtt{B})] : \mathsf{fresh}\ \mathcal{X}_4.\mathsf{send}(\mathsf{insec}, \mathtt{scrypt}(\mathcal{X}_3, \mathcal{X}_4)$$

and the instance $\sigma = [\mathtt{A} \mapsto \mathtt{a}, \mathtt{B} \mapsto \mathsf{i}]$, we get the countably many operational strands $[\mathcal{X}_1 \mapsto \mathtt{a}, \mathcal{X}_2 \mapsto \mathsf{i}, \mathcal{X}_3 \mapsto \mathtt{shk}(\mathtt{a}, \mathsf{i})] : \mathsf{fresh}\ \mathcal{X}_4.\mathsf{send}(\mathsf{insec}, \mathtt{scrypt}(\mathtt{shk}(\mathtt{a}, \mathsf{i}), X_4)).\mathsf{finished}(n).0$ for each $n \in \mathbb{N}$. All remaining variables in the instantiated operational strands represent freshly created values and (parts of) received messages.

Note that here we have created only the instances for the honest agents (because of the side condition $\sigma(R_i) \neq \mathsf{i}$); this is so since the behavior of the honest agents is subsumed by the abilities of the intruder when given the appropriate knowledge of the role in all instances where he plays the role.[4] With $\mathcal{M}_0$ we therefore define the initial knowledge that the intruder needs to play in all roles under his real name. Here we model the *intruder knowledge* simply as a *set* of messages (rather than a substitution M as for honest agents) as for the standard Dolev-Yao intruder deduction, we do not need labels (and we do not consider notions like behavioral equivalence here). Accordingly, let $\vdash'$ denote the standard unlabeled intruder deduction on unlabeled messages defined as follows: $ul(\mathcal{M}) \vdash' t$ iff $\mathcal{M} \vdash t^l$ for some l.

The transition relation The transition relation $\Longrightarrow$ is defined as the least relation closed under the following rules:

$$\text{T1} \quad (\{\mathsf{send}(\mathsf{insec}, t).\mathit{rest}\} \cup \mathcal{S}; \mathcal{M}; E) \Longrightarrow (\{\mathit{rest}\} \cup \mathcal{S}; \mathcal{M} \cup \{t\}; E)$$

$$\text{T2} \quad (\{\mathsf{receive}(\mathsf{insec}, t).\mathit{rest}\} \cup \mathcal{S}; \mathcal{M}; E) \Longrightarrow (\{\sigma(\mathit{rest})\} \cup \mathcal{S}; \mathcal{M}; E)$$
$$\text{for any substitution } \sigma \text{ such that } \mathcal{M} \vdash' \sigma(t)$$

$$\text{T3} \quad (\{\mathsf{event}(t).\mathit{rest}\} \cup \mathcal{S}; \mathcal{M}; E) \Longrightarrow (\{\mathit{rest}\} \cup \mathcal{S}; \mathcal{M}; E \cup \{\mathsf{event}(t)\})$$

$$\text{T4} \quad (\{s \doteq t.\mathit{rest}\} \cup \mathcal{S}; \mathcal{M}; E) \Longrightarrow (\{\mathit{rest}\} \cup \mathcal{S}; \mathcal{M}; E) \quad \text{if } s \approx t$$

$$\text{T5} \quad (\{\mathsf{fresh}\ X_i.\mathit{rest}\} \cup \mathcal{S}; \mathcal{M}; E) \Longrightarrow (\{\sigma(\mathit{rest})\} \cup \mathcal{S}; \mathcal{M}; E)$$
$$\text{where } \sigma = [\mathcal{X}_i \mapsto c] \text{ and } c \text{ is a fresh constant}$$

$$\text{T6} \quad (\{\mathcal{X}_i := t.\mathit{rest}\} \cup \mathcal{S}; \mathcal{M}; E) \Longrightarrow (\{\sigma(\mathit{rest})\} \cup \mathcal{S}; \mathcal{M}; E)$$
$$\text{where } \sigma = [\mathcal{X}_i \mapsto t]$$

$$\text{T7} \quad (\{0\} \cup \mathcal{S}; \mathcal{M}; E) \Longrightarrow (\mathcal{S}; \mathcal{M}; E)$$

The rules T1 and T2 handle the sending and receiving over an insecure channel: we add every sent message t to the intruder knowledge; for an agent who wants to receive a message of the *form* t (note that t may contain variables that are bound in this step), the intruder can use any instance $\sigma(t)$ that he can derive from his knowledge and we apply σ to the rest of the strand, i.e., instantiating all variables that have been bound in this step. We have only discussed the standard case of insecure channels here, other kinds of channels can be defined as in the ideal channel model of [MV09a].

[4] In fact, we define here the semantics of operational strands using a standard Dolev-Yao style intruder deduction relation; stronger models could be employed, we just require that the intruder can at least perform the actions that honest agents can, i.e., encryption and decryption with known keys and the like.

The rule T3, simply adds the $\mathsf{event}(t)$ to the set of events E indicating that $\mathsf{event}(t)$ has occurred. T4 says that if we face the check $s \doteq t$ then we proceed if the check is met. In the rule T5, we handle the creation of a fresh value $\mathcal{X}_i$ by substituting all the next occurrences of $\mathcal{X}_i$ by a fresh constant c. A fresh constant is simply a constant that does not occur elsewhere, we assume that each agent (including the intruder) has an infinite reservoir of fresh constants. T6 removes the strand-macro $\mathcal{X}_i := t$ by replacing $\mathcal{X}_i$ with what it abbreviates, i.e., the term t. The last rule T7 handles the end of the strand.

Note that the following invariant holds over all transitions: the intruder knowledge is a set of ground terms, all strands are closed, and all terms that the intruder can derive and send are thus also ground.

3.6 Summary

In this chapter, we defined the semantics of SPS in a concise and simple way that works for an arbitrary theory. Our high-level semantics $[\![\cdot]\!]_H$ gives a mathematically succinct and uniform definition of Alice-and-Bob notation following from a few general principles, and at the same time it supports an arbitrary set of operators and algebraic properties. The succinctness and generality are, in our opinion, a strong argument for this semantics as a standard. As $[\![\cdot]\!]_H$ entails problems that are not recursively computable in general, we defined the low-level semantics $[\![\cdot]\!]_L$ for a particular theory and proved its correctness with respect to $[\![\cdot]\!]_H$, i.e., we proved that $[\![\cdot]\!]_L$ implements $[\![\cdot]\!]_H$ and that both semanticses produce equivalent operational strands given then same plain strand. In the next chapter we define translators from operational strands to implementations and formal models.

Chapter 4

Beyond the Semantics

We now come to the "last mile" of the translation: to translate operational strands into actual implementations and into formal models for automated verification. We implemented these translations (and others) in the SPS compiler[1] that we depict in Figure 4.1. We give the details of the "JEE" later in this chapter, and the details of "APCC" in Chapter 9. As target languages, we have here JavaScript[2] for protocol implementations and Applied π for the formal model.

One can easily see a very close correspondence between the two translations: roughly, they both use the same operators in the same way, only in the formal model they are function symbols in an "abstract" term algebra, whereas in the implementation they are *corresponding* API calls. It is one of the contributions of this work to achieve such a close correspondence. While the use of crypto-APIs is of course standard, our notion of formats extends this API idea also to the non-cryptographic operations: all the technical details of parsing and pretty-printing are hidden in the classes for the given formats. Of course, just like the crypto-API, also the "non-crypto-APIs" require a robust implementation (that does not suffer from buffer overflows, for instance), but we want to argue that our setup with APIs is a suitable way to "cut the cake".

[1] available at http://www2.imm.dtu.dk/~samo/SPS.zip

[2] One may argue that JavaScript is not suitable for implementing security protocols, but in fact, using systematic mechanisms such as our formats, we can produce robust implementations that do not suffer from type flaw attacks, for instance.

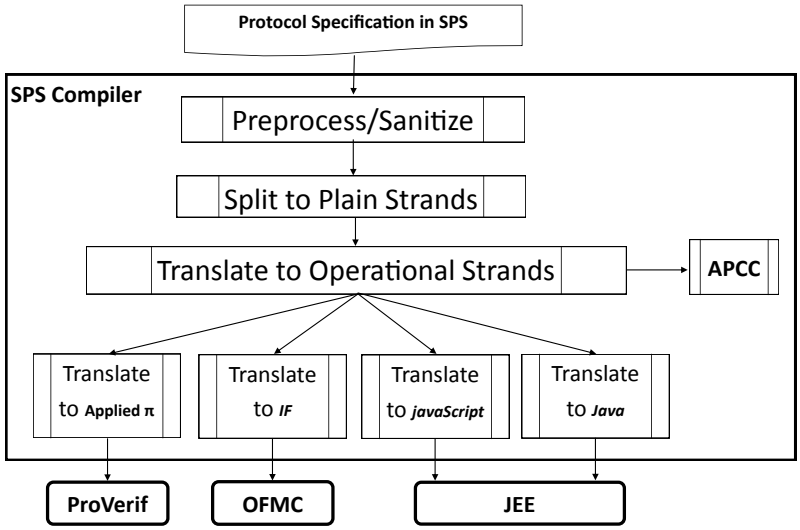


Figure 4.1: The SPS Compiler

The close correspondence allows us to argue that there is no systematic discrepancy between formal model and implementation, if the function symbols have the corresponding meaning—but that is indeed subtle. Comparing the translation with the input strand of Figure 3.1(right), there are only two significant differences: all the explicit verifiers of the strands are removed (as we explain later) and the implementation does not contain events; besides that, the translation is mainly adapting to the syntax of the target language.

We proceed by giving the translations to JavaScript as a representative for implementations, and Applied π as a representative for formal models.

4.1 Translation to JavaScript

We define the function $[\![\cdot]\!]_{JS}$ that translates from operational strand to JavaScript code. In the definition below we use + for string concatenation.

where: $head(M : steps) =$ "`void proc_`"$+own(M : steps)+$"("$+par(M)+$"){",

$$
\begin{array}{ll}
[\![M : steps]\!]_{JS} & = head(M : steps) + [\![steps]\!]_J \\
[\![\mathsf{receive}(ch, \mathcal{X}_i).rest]\!]_J & = \text{“}\texttt{var }\text{”} + \mathcal{X}_i + \text{“} = ch.\mathsf{receive}();\text{”} + [\![rest]\!]_J \\
[\![\mathsf{send}(ch, l).rest]\!]_J & = \text{“}ch.\mathsf{send}(\text{”} + l + \text{“});\text{”} + [\![rest]\!]_J \\
[\![\mathsf{fresh}\ \mathcal{X}_i.rest]\!]_J & = \text{“}\texttt{var }\text{”} + \mathcal{X}_i + \text{“} = \texttt{genNumber}();\text{”} + [\![rest]\!]_J \\
[\![\mathsf{verify}_{\mathtt{f}}(l) \doteq \top.rest]\!]_J & = \texttt{f} + \text{“ ”} + l + \text{“}a = \texttt{new }\text{”} + \texttt{f} + \text{“}(\text{”} + l + \text{“});\text{”} \\
 & \quad + [\![rest]\!]_J \\
[\![\mathcal{X}_i := \mathsf{get}_{i,\mathtt{f}}(l).rest]\!]_J & = \text{“}\texttt{var }\text{”} + \mathcal{X}_i + \text{“} = \text{”} + l + \text{“}a.get\text{”} + i + \text{“}();\text{”} \\
 & \quad + [\![rest]\!]_J \\
[\![\mathcal{X}_i := t.rest]\!]_J & = \text{“}\texttt{var }\text{”} + \mathcal{X}_i + \text{“} = \text{”} + t + \text{“};\text{”} + \ [\![rest]\!]_J \\
[\![t \doteq \top.rest]\!]_J & = [\![rest]\!]_J \\
[\![t \doteq s.rest]\!]_J & = \text{“}\texttt{if}(\text{”} + t + \text{“! = ”} + s + \text{“) }\texttt{error}(); + [\![rest]\!]_J \\
[\![0]\!]_J & = \text{“}\}\text{”}
\end{array}
$$

$own(M : steps)$ is the agent that owns the strand $M : steps$, and $par(M)$ is the knowledge M formed as a list of parameters, i.e., a comma separated list of the label variables of the knowledge M. We add to this list a channel object ch given as an additional parameter that the code uses to send and receive messages as we explain later. In a nutshell, the first rule gives the header of the JavaScript code that we want to generate from the operational strand S. For example, let S be the operational strand given in Figure 3.1(right), then $head(S) = \texttt{function proc\_A}(\texttt{X1}, \texttt{X2}, \texttt{X3}, \texttt{X4}, \texttt{ch})\{$. The left bracket that we have at the end starts the function.

In the receive rule, we declare a new variable $\mathcal{X}_i$ and assign to it the value received from the channel ch via the method receive(), i.e., the value obtained from ch.receive() is assigned to that variable $\mathcal{X}_i$. Next, the send rule uses the method send() to send the term l over the channel ch . Recall that l is a recipe that tells how to construct some value in reference to the given parameters (initial knowledge), received messages, or derived messages. The fresh $\mathcal{X}_i$ rule simply translates to a creation of a new value $\mathcal{X}_i$.

The forth rule of $[\![\cdot]\!]_J$ handles a special case of checks, namely the case of a format verifier ($\mathsf{verify}_{\mathtt{f}}(l) \doteq \top$) where l represents what is supposed to be a “serialized” `f` object, i.e., a byte string that is supposed to be parsed as an object of type `f`. In this case, we create an object of that format where the name of the object is obtained simply by appending the letter ‘a’ to the string name l (there is no significance in choosing the letter ‘a’, it is just that we need to distinguish between the byte string called l and the new object that we need to create when we parse l, so we called the object $l + \text{“}a\text{”}$). The reason behind the creation of an object is of twofold. First, we need this object for later use (in obtaining the different fields of the formatted message as we see in the next rule); for that we cannot directly use the byte string l. Second, we need to verify that the string l is indeed of the format `f`. This verification is not explicitly done,

instead it is left to the constructor that maps l to an object of type `f`. The constructor here is basically a format parser. The next rule is dedicated to macros in which format getters are involved. Recall that by a format getter we mean the format methods that obtain different fields of a format object. This is achieved simply by calling the `get` method of the format object that we created when we encountered the format verifier ($\texttt{verify}_{\texttt{f}}(l)$). Here we rely on the fact that our model generates a format verifier before decomposing a format. Note that this case is a special case of macros, the next rule ($[\![X_i := s.rest]\!]_J$) handles the other cases of macros. When we have the macro $\mathcal{X}_i := t$, we simply create a new variable $\mathcal{X}_i$ and give it the value t.

In the seventh rule we handle another special case of checks, namely the case with operator verifiers except the format verifiers (we already handled format verifiers in the third rule). The operator verifiers that we have in SPS as we discussed earlier are {`vcrypt`, `vsign`, `vscrypt`}. In this case, we do not produce any JavaScript code; simply because the verification is left to the deconstructors {`dcrypt`, `open`, `dscrypt`}; since they implement a verification mechanism, e.g., the decryption will raise an exception if it failed to decrypt a supposedly encrypted message. Here, we rely on the fact that our model calls the decryption step immediately after the verification (cf. Table 3.2), so we delay the verification to the decryption that implements it. The eighth rule is for the other cases of checks, namely when we require that two terms (none of them is $\top$) are equal. We translate this check into an if-statement, i.e., if the two labels of the check are not equal then we arise an error. The last rule handles the end of the operational strand that we translate to a right bracket (}) closing the left bracket we opened in the very first rule (by calling $head(\cdot)$). Figure 4.3 shows this translation for the role `A` of our example in Figure 3.1(right).

We designed SPS in FutureID project to enable a service called the Universal Authentication Service (UAS) [Fut13b] to execute arbitrary authentication protocols.UAS consists of an execution environment (called the "Job Execution Environment" and depicted in Figure 4.2 and specified in [Fut14]) that provides libraries of cryptographic functions to link abstract code calls to concrete functions e.g., `crypt` to RSA or `scrypt` to AES. This link is maintained by a set of configuration files for each SPS specification (or protocol). Moreover, UAS provides required format classes that a protocol needs. This setting makes our generated code inter-operable with any given message formats (any mechanism of message parsing) and with cryptographic libraries. This enables us to model real-world protocols like EAC used in German eID and TLS.

Crypto API. We of course rely on the execution environment to have suitable implementations of the cryptographic primitives, e.g., the `exp` operator will

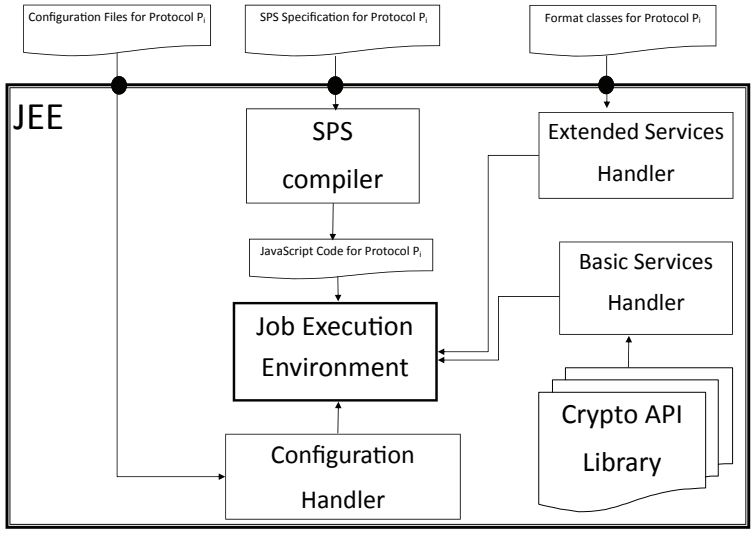


Figure 4.2: The Job Execution Environment

in fact be mapped to elliptic curve cryptography. We assume that the call $\texttt{dscrypt}(k, m)$ will fail (aborting execution) if m is not a message encrypted with key k. This is why we do not include verifier checks in this translation. For simplicity, we omitted the optional annotation of primitives with the precise algorithm and key length (that is only necessary when using different ones in the same protocol).

Formats. The notion of formats allows us to integrate the actual message formats of real-world protocols like EAC and TLS. Similar to the cryptographic operators, we also rely on what we like to call a *non-crypto API* that is an implementation of non-cryptographic operators: for each format declared in the specification, we require a Java class that implements it. Each format class must have the following:

1. A private member variable for each of its fields.

2. A constructor that gets values for member variables. It corresponds to our abstract function symbol in SPS.

3. A constructor that gets a string in concrete syntax and parses it throwing an exception if required elements are missing or the message is not well-formed; i.e., it preforms the check that corresponds to the format that $\mathsf{verify}_{\mathtt{f}}$ function represents in SPS.

4. A method `encode` to "pretty-print" the object into a concrete string.

   The latter two are respectively called *serializer* and *de-serializer* in protocol implementation slang.

5. A `get` function for every element of the object; this corresponds to $\mathtt{get}_{i,\mathtt{f}}$.

For the example format `f1`, we have the class `f1` must have three member variables of type byte string to represent the three fields of the form (as raw data). It must have two constructors: the first takes three byte strings as input and just stores them in the member variables (cf. the first `new f1` in the example), the second takes a single byte string and tries to parse it as format `f1`, and this may fail (cf. the second `new f1` in the example). Further, we have the `get`i() functions to obtain the i-th field and `encode`() to output a byte string. For a more detailed discussion of formats of EAC, TLS and other protocols see Chapter 5.

```
function proc_A(X1,X2,X3,X4,ch){
 Number  X5 = genNumber();
 ch.send(scrypt(X3,new f1(X1,X2,
   exp(X4,X5)).encode()));
 var X6 = ch.receive();
 var X7 = dscrypt(X3, X6);
 f1 X7a = new f1(X7);
 var X8 = X7a.get1();
 var X9 = X7a.get2();
 var X10 = X7a.get3();
 if(X8 != X2) error();
 if(X9 != X1) error();
 Number  X11 = genNumber();
 ch.send(scrypt(exp(X10,X5),
    new f2(X11).encode()));

}
```

```
let proc_A(x1,x2,x3,x4:bitstring,ch:Chann)=
  new x5:bitstring;
  out(ch,scrypt(x3,f1(x1,x2,
    exp(x4,x5))));
  in(ch,x6:bitstring);
  let x7:bitstring = dscrypt(x3,x6) in

  let x8:bitstring = f1get1(x7) in
  let x9:bitstring = f1get2(x7) in
  let x10:bitstring = f1get3(x7) in
  if(x8 = x2) then
  if(x9 = x1) then
  new x11:bitstring;
  out(ch,scrypt(exp(x10,x5),
    f2(x11)));
  event secret(x1,x2,x11);
  0.
```

Figure 4.3: Translation to JavaScript and Applied π of the role A

4.2 Translating to Applied π

Proverif allows for the verification of protocols for an unbounded number of sessions. Proverif uses abstraction and produces sound security proofs, i.e., the absence of attacks in the given protocol model. However, the abstraction may lead to false positive attacks due to the over-approximation, i.e., some attacks may be unrealistic. Proverif accepts as input specifications in Applied π [AF01].

In order to connect SPS to ProVerif, we implemented within the SPS compiler a translation to Applied π code. The generated code is composed of three parts: declarations, processes and a global process. Now we give some details using the Applied π generated code for our example protocol using the SPS compiler.

The Declaration Part

The first part of the generated Applied π code is the declaration part, in which we declare all identifiers and functions to be used in the rest of the code. Listing 4.1 shows a chunk of this part, where one can find the declaration of the following:

- A channel `ch` (Line 2) that agents use to exchange message.
- The intruder name (Line 3).
- The mappings that we use in the protocol, like `shk` (Line 4). Note that it has the property of being `private`, i.e., agents (including the intruder) cannot use it to compose messages.
- The used functions such as the symmetric encryption function `scrypt` (Line 15) followed by its decryption property; namely that if the key is known then the message is also known. Proverif considers all functions to be public unless they are followed by `[private]` as in the mapping `shk`, thus another property of `scrypt` is that it is public (all agents including the intruder can use it to compose messages).
- The used formats, e.g., `f1`(Line 5) followed its properties (Lines 6, 7 and 8), i.e., if one has a formatted message then he can get the fields of that formats (decompose the format). Recall that formats are also public.
- Finally, the goals. (Line 21) is a query for Proverif to answer if the attacker can get the enclosed secret `PayloadAB` that is declared to be private in the previous line.

```
1 (*Auto-Generated by SPS*)
2 free ch: channel.
3 free i:bitstring.
4 fun shk(bitstring,bitstring):bitstring [private].
5 fun f1(bitstring,bitstring,bitstring):bitstring.
6  reduc forall a,b,c:bitstring; f1get1(f1(a,b,c))=a.
7  reduc forall a,b,c:bitstring; f1get2(f1(a,b,c))=b.
8  reduc forall a,b,c:bitstring; f1get3(f1(a,b,c))=c.
9 const g: bitstring.
10 fun exp(bitstring,bitstring):bitstring.
11  equation forall  x, y: bitstring;
12     exp(exp(g,x),y)=exp(exp(g,y),x).
13 fun f2(bitstring):bitstring.
14  reduc forall a:bitstring;f2get1(f2(a))=a.
15 fun scrypt(bitstring,bitstring):bitstring.
16  reduc forall k: bitstring, m: bitstring;
17     sdecrypt(k,scrypt(k,m))=m.
18
19 (* END OF FUNCSansPROPS*)
20
21 free PayloadAB:bitstring[private].
22
23 query attacker(PayloadAB).
```

Listing 4.1: Declaration part of the example protocol in Applied π

Note that in this part we handled the most subtle problem: the algebraic properties of the cryptographic and non-cryptographic operators. We can express cancellation, e.g.,

$$\texttt{reduc forall m,k : bitstring; dscrypt(k, scrypt(k, m))} = \texttt{m}.$$

Note that directly formulating the equations for `exp` and `mult` will cause non-termination in ProVerif [BS11, BSC14]. Thus, for standard Diffie-Hellman, we use the property in lines 9–12 that is a sound restriction that ProVerif can handle [KT09, Möd11]:

$$\texttt{equation forall x, y : bitstring; exp(exp(g, x), y)} = \texttt{exp(exp(g, y), x)}\,.$$

The translator can give a warning when the SPS specification is outside the fragment for which the soundness result holds.

The Processes Part

The second part of the generated Applied π code consists of the processes of the protocol participants, i.e., `A` and `B` in our example protocol. These processes contain all actions each participant performs such as sending and receiving messages, performing checks and the events he issues. The events are used later to verify goals. Here, we present $[\![\cdot]\!]_\pi$ (with $[\![\cdot]\!]_{\pi_0}$ as the initial case) that translates an operational strand to an Applied π calculus process. We use the syntax provided in [RS11]. Note that the semantics of operational strands is actually similar to a process calculus and this translation to it is mainly a matter of pretty-printing. We define $[\![\cdot]\!]_\pi$ as follows (+ denotes string concatenation):

$$[\![M : steps]\!]_{\pi_0} = \text{“}\texttt{let process}\text{“} + own(M : steps) + \text{“(”} + par(M) + \text{“)=”} + [\![strand]\!]_\pi$$

where: $own(M : steps)$ is the name of the agent that owns the strand $strand$, and $par(M)$ is a list of the process parameters derived from its initial knowledge M (an example is given below)

$$\begin{array}{lcl}
[\![\mathsf{send}(ch, l).rest]\!]_\pi & = & \text{“}\texttt{out}\text{(”} + ch + \text{“,”} + \text{l} + \text{“);”} + [\![rest]\!]_\pi \\
[\![\mathsf{receive}(ch, l).rest]\!]_\pi & = & \text{“}\texttt{in}\text{(”} + ch + \text{“,”} + \text{l} + \text{“);”} + [\![rest]\!]_\pi \\
[\![\mathsf{fresh}\ l.rest]\!]_\pi & = & \text{“}\texttt{new}\text{ ”} + \text{l} + \text{“:}\texttt{bitstring}\text{;”} + [\![\text{rest}]\!]_\pi \\
[\![x := t.rest]\!]_\pi & = & \text{“}\texttt{let}\text{ ”} + \text{x} + \text{“=”} + \text{t} + \text{“}\texttt{in}\text{ ”} + [\![rest]\!]_\pi \\
[\![t \doteq \top.rest]\!]_\pi & = & [\![rest]\!]_\pi \\
[\![s \doteq t.rest]\!]_\pi & = & \text{“}\texttt{if}\text{(”} + \text{s} + \text{“=”} + \text{t “)}\ \texttt{then}\text{ ”} + [\![rest]\!]_\pi \\
[\![\mathsf{event}(t).rest]\!]_\pi & = & \text{“}\mathsf{event}(t)\text{;”} + [\![rest]\!]_\pi \\
[\![0]\!]_\pi & = & \text{“0.”}
\end{array}$$

The first rule declares the agent's process by giving it a name and parameterizing it over the initial knowledge of the agent. For example, Let $S^{\mathtt{A}} = M^{\mathtt{A}} : steps^{\mathtt{A}}$ be the strand shown in our example in Figure 3.1(right), then $own(S^{\mathtt{A}}) = \mathtt{A}$, and $par(M^{\mathtt{A}}) = \mathtt{x1}, \mathtt{x2}, \mathtt{x3}, \mathtt{x4} : \mathtt{bitstring}$, so the process will be `process` as shown in the first line of Listing 4.2. The second and the third rules deal with the sending and receiving of messages over a channel ch. The forth rule deals with the creation of a fresh value, and the fifth rule covers the macro case of a strand and how it is translated in Applied π code. The sixth and the seventh rules deal with the checks. Note that we simply ignore checks with $\top$ on one side; since such checks are implicitly performed by the next destruction step. For example, in the translation of $S^{\mathtt{A}}$, we ignore $\mathsf{vscrypt}(\mathcal{X}_3, \mathcal{X}_6) \doteq \top$ as it is followed by $\mathcal{X}_7 := \mathtt{dscrypt}(\mathcal{X}_3, \mathcal{X}_6)$, which according to the property (`reduc forall m,k : bitstring; dscrypt(k,scrypt(k,m)) = m.`) will not be decrypted $\mathcal{X}_6$ unless it is a valid encrypted message and $\mathcal{X}_3$ is a valid encryption key. The eighth rule pretty-prints the event $\mathsf{event}(t)$ in the process and the last

rule ends the strand. The result of applying $[\![\cdot]\!]_\pi$ to the operational strand of the agents `A` and `B` is shown in Listing 4.2.

```
(*Agent A process in protocol:Example  is *)
let processA(x1:bitstring,x2:bitstring,x3:bitstring,x4
    :bitstring) =
new x5:bitstring;

out(ch, scrypt(x3, f1(x1, x2, exp(x4, x5))));
in(ch, x6:bitstring);
let x7:bitstring = sdecrypt(x3, x6) in
let x8:bitstring = f1get1(x7) in
let x9:bitstring = f1get2(x7) in
let x10:bitstring = f1get3(x7) in
if(x8 = x2) then
if(x9 = x1) then
new x11:bitstring;

out(ch, scrypt(exp(x10, x5), f2(x11)));
if(x2<> i) then
out(ch, scrypt(x11,PayloadAB));0.
(* ------------------------- *)
        (*EOP*)
(*Agent B process in protocol:Example  is *)
let processB(x1:bitstring,x2:bitstring,x3:bitstring,x4
    :bitstring) =
in(ch, x5:bitstring);
let x6:bitstring = sdecrypt(x3, x5) in
let x7:bitstring = f1get1(x6) in
let x8:bitstring = f1get2(x6) in
let x9:bitstring = f1get3(x6) in
if(x7 = x1) then
if(x8 = x2) then
new x10:bitstring;
out(ch, scrypt(x3, f1(x2, x1, exp(x4, x10))));
in(ch, x11:bitstring);
let x12:bitstring = sdecrypt(exp(x9, x10), x11) in
let x13:bitstring = f2get1(x12) in
if(x1<> i) then
out(ch, scrypt(x13,PayloadAB));0.
```

Listing 4.2: The processes part in Applied π

This part also handles a subtlety of algebraic properties, i.e., during the verification process of ProVerif, where processes get translated into Horn clauses, ProVerif encodes the destructors (declared in the declaration part and explained above) into pattern matching—in the Horn clauses occur no destructors or verifiers. This transformation corresponds to an implicit verifier: in our example, the `let x6` clause in Line 23 will fail if the message `x5` is not of the form `scrypt(x3, ·)`. Moreover, the subsequent three "let" clauses will fail if `x5` is not formatted with `f1`. Thus, the ProVerif translation does not have verifiers either.

The Global Process Part

The third part is the "global" process that instantiates the above processes with their initial knowledges and allows the intruder to instantiate some of these process to model the dishonest agents behavior. In the generated code, we formulate all possible instantiations of the protocol: every role can be played by any agent, including the intruder, and we want to allow for any number of sessions of the protocol in parallel. It is not trivial to specify this manually, but the SPS compiler offers a systematic way to generate the instantiation. Recall that the initial knowledge of each role in the SPS specification can only contain variables of type `Agent` and long-term keys have to be specified using functions like `shk`. This allows us to instantiate the knowledge for any value of the role variables. For our example, we have the code in Listing 4.3 (where `ch` is the insecure channel that we declared previously):

```
process

!new x:bitstring;
out(ch, x)|
!in(ch,(b:bitstring));
processA(x, b, shk(x, b), g) |
out(ch,(i, b, shk(i, b), g))|
!in(ch,(a:bitstring));
processB(a, x, shk(a, x), g) |
out(ch,(a, i, shk(a, i), g))
```

Listing 4.3: The "global" process part

The first replication operator (!) generates an unbounded number of honest agent names (in variable `x`) that are broadcast on `ch`. Then we generate an unbounded number of instances of `processA` for each `x` and each name `b` that we receive from the public channel (thus, the intruder can choose who will play

role B). We also output on ch the initial knowledge that the intruder needs for playing role A under his real name i. The last two lines are similar but for role B. The full generated code is shown below (in Listing 4.4).

```
(*Auto-Generated by SPS*)
free ch: channel.
free i:bitstring.
fun shk(bitstring,bitstring):bitstring [private].
fun f1(bitstring,bitstring,bitstring):bitstring.
    reduc forall a:bitstring,b:bitstring,c:bitstring;
        f1get1(f1(a,b,c))=a.
    reduc forall a:bitstring,b:bitstring,c:bitstring;
        f1get2(f1(a,b,c))=b.
    reduc forall a:bitstring,b:bitstring,c:bitstring;
        f1get3(f1(a,b,c))=c.
const g: bitstring.
fun exp(bitstring,bitstring):bitstring.
    equation forall  x: bitstring, y: bitstring; exp(
        exp(g,x),y)=exp(exp(g,y),x).
fun f2(bitstring):bitstring.
    reduc forall a:bitstring;f2get1(f2(a))=a.
fun scrypt(bitstring,bitstring):bitstring.
    reduc forall k: bitstring, m: bitstring;  sdecrypt
        (k,scrypt(k,m))=m.

(* END OF FUNCSansPROPS*)

free PayloadAB:bitstring[private].

query attacker(PayloadAB).

(*Agent A process in protocol:Example  is *)
let processA(x1:bitstring,x2:bitstring,x3:bitstring,x4
    :bitstring) =
new x5:bitstring;

out(ch, scrypt(x3, f1(x1, x2, exp(x4, x5))));
in(ch, x6:bitstring);
let x7:bitstring = sdecrypt(x3, x6) in
let x8:bitstring = f1get1(x7) in
let x9:bitstring = f1get2(x7) in
```

```
32 let x10:bitstring = f1get3(x7) in
33 if(x8 = x2) then
34 if(x9 = x1) then
35 new x11:bitstring;
36
37 out(ch, scrypt(exp(x10, x5), f2(x11)));
38 if(x2<> i) then
39 out(ch, scrypt(x11,PayloadAB));0.
40 (* -------------------------- *)
41         (*EOP*)
42 (*Agent B process in protocol:Example  is *)
43 let processB(x1:bitstring,x2:bitstring,x3:bitstring,x4
       :bitstring) =
44 in(ch, x5:bitstring);
45 let x6:bitstring = sdecrypt(x3, x5) in
46 let x7:bitstring = f1get1(x6) in
47 let x8:bitstring = f1get2(x6) in
48 let x9:bitstring = f1get3(x6) in
49 if(x7 = x1) then
50 if(x8 = x2) then
51 new x10:bitstring;
52
53 out(ch, scrypt(x3, f1(x2, x1, exp(x4, x10))));
54 in(ch, x11:bitstring);
55 let x12:bitstring = sdecrypt(exp(x9, x10), x11) in
56 let x13:bitstring = f2get1(x12) in
57 if(x1<> i) then
58 out(ch, scrypt(x13,PayloadAB));0.
59 (* -------------------------- *)
60         (*EOP*)
61
62 process
63
64 !new x:bitstring;
65 out(ch, x)|
66 !in(ch,(b:bitstring));
67 processA(x, b, shk(x, b), g) |
68 out(ch,(i, b, shk(i, b), g))|
69 !in(ch,(a:bitstring));
70 processB(a, x, shk(a, x), g) |
```

```
71 out(ch,(a, i, shk(a, i), g))
```

Listing 4.4: The Auto-generated Applied π code by SPS compiler for the running example

4.3 Summary

The SPS compiler enables the formal verification of protocols via the connection to several back-end tools such as the static analysis tool Proverif and the model checker OFMC. Moreover, SPS generates JavaScript implementations that are based on robust APIs for cryptographic and formats. Both the formal models and the implementations are derived from the same operational strands. In this chapter, we explained two translations from operational strands in detail. First, we explained our translation to JavaScript giving some contextual details about the SPS compiler in general and how it fits within the FutureID project. Then, we explained our translation to Applied π [AF01] as representatives for formal models. We detailed how each part of the generated code is handled. We think that the systematic instantiation of Applied π processes tackles a subtlety that many non-expert users may face, namely that such users may make mistakes in the instantiation of processes. In the next chapter, we consider in details some real-world protocols as case studies, namely: EAC [Ger08], PACE [Fed12], TLS [DR08], and two of the ISO/IEC 9798 [Int99] standard authentication protocols.

Chapter 5

Case Studies

We introduce the SPS files for some selected protocols that are highly relevant to electronic identity systems (eID) in general and the FutureID project in particular. The selected protocols are: (1) the Extended Access Control protocol (EAC) that is used in the European e-passports to protect sensible data. (2) The Password Authenticated Connection Establishment protocol (PACE) that establishes a secure channel between an eID card chip and a terminal. PACE is the recommended first step in EAC. (3) The Transport Security Layer protocol (TLS) that is one of the most widely used protocols to establish secure channels between a client and a server. (4) A group of ISO/IEC 9798 standard authentication protocols. Next in Section 5.1, we present each of the selected protocols in a separate section, but following the same structure for all protocols as we explain: for each of the selected protocols, we introduce it briefly, then explain in details its specification in SPS. After that we give a tabular summary for the results of our test suite.

5.1 Case Studies Structure

In order to improve the readability of this chapter, we follow the same structure in presenting the selected protocols. We introduce each protocol in a separate section that has the following subsections:

- An introduction, in which we give a brief description of the protocol.
- The SPS specification of the protocol, in which we explain how we modeled different aspects of the protocol. As a natural choice, we present the protocol's SPS specification following its structure as explained in details in Chapter 2.
- The protocol formats, in which we discuss the message formats of the protocol.
- The analysis results, in which we discuss the formal verification of the protocol. We also present the most important aspects of the auto-generated code. Then we summarize the results obtained from running the back-end tools (ProVerif and OFMC) on the auto-generated code of the protocol.

5.2 EAC

The Extended Access Control (EAC) protocol [Ger08] is used in eID cards (ePassports) to secure sensitive data like fingerprints. EAC is a two-party protocol that aims to provide a mutual authentication between an eID card and a terminal. The protocol takes place in the environment of the German ID card. Through EAC (1) a terminal (Proximity Coupling Device, `PCD`) authenticates itself to the ID card (Proximity Integrated Circuit Card, `PICC`) to get access to the sensitive data stored on the card and (2) the `PICC` proves its authenticity to the `PCD`. In brief, EAC works as follows: (1) `PCD` sends his certificate (issued by a certificate authority) to `PICC`, (2) `PICC` replies with a freshly generated random number say R_1. Then, (3) `PCD` sends back his Diffie-Hellman half key, and (4) `PICC` replies with another freshly generated random number say R_2. Now, (5) `PCD` signs R_1 and his Diffie-Hellman half key, and sends the signature to `PICC`. Finally, (6) `PICC` sends his Diffie-Hellman half key signed by a certificate authority that `PCD` accepts. Both `PICC` and `PCD` can then construct a shared authenticated key.

5.2.1 EAC in SPS

In Listing 5.1, we give the complete specification of EAC followed by a detailed explanation for this specification.

```
1 Protocol: EAC
2 Types:
3  Agent PCD, PICC, ca;
```

```
Number g, certDesc, rChat, oChat, auxData, noCert, cHAT, cAR, eFC, idPICC,
    null,
   handle,  didName;
Formats:
eac1input(Msg, Msg, Msg, Number, Number, Number, Number);
eac1output(Number, Number, Number, Number, Number);
eac2input(Msg, Msg, Number, Number, Number);
eac2output(Number, Msg, Msg, Number);
eacadditionalinput(Msg, Msg, Msg);
certForm(Agent, PublicKey);
x59d(Agent, Number, PublicKey);
Knowledge:
PCD:  PCD, sign(inv(pk(ca)),certForm(PCD,pk(PCD))), pk(PCD), pk(ca),
   inv(pk(PCD)), g, handle, didName, certDesc, rChat, oChat, auxData, noCert
      , null;
PICC: PICC, sign(inv(pk(ca)),certForm(PICC,exp(g,sk(PICC)))), pk(ca),
   sk(PICC), g, cHAT, cAR, eFC, idPICC, null;
 where PCD!=ca, PICC !=ca;
Actions:
[PCD]*->*[PICC]: eac1input(handle,  didName, sign(inv(pk(ca)), certForm(PCD,
    pk(PCD))), certDesc, rChat, oChat, auxData)

PICC: Number  RC
[PICC]*->*[PCD]: eac1output(cHAT, cAR, eFC, idPICC, RC)

PCD: Number X
let PK_PCD=exp(g,X)
[PCD]*->*[PICC]: eac2input(handle,  didName, noCert, PK_PCD, null)

PICC: Number Rpicc
[PICC]*->*[PCD]: eac2output(Rpicc, null, null, null)

[PCD]*->*[PICC]: eacadditionalinput(handle,  didName, sign(inv(pk(PCD)),
    x59d(PICC, Rpicc, PK_PCD)))

PICC: Number Rmac
let PK_PICC=exp(g,sk(PICC))
let DHKey= exp(PK_PCD,sk(PICC))
[PICC]*->*[PCD]: eac2output(null, sign(inv(pk(ca)),certForm(PICC,PK_PICC)),
    mac(hash(DHKey, Rmac), PK_PCD), Rmac)

Goals:
PICC authenticates PCD on DHKey
PCD authenticates PICC on DHKey
DHKey secret of PICC, PCD
```

Listing 5.1: EAC in SPS

Types: In EAC, we have three agents: (1) `PCD` represents a terminal (Proximity Coupling Device), (2) `PICC` represents an eID card (Proximity Integrated Circuit Card), and (3) `ca` that represents a trusted certificate authority that issued certificates for the other two agents. We also declare a number `g` to represent a base to use it later for Diffie-Hellman key agreement. We assume that g is a primitive root modulo some prime p that

both agents agreed upon; we later use exp(g, X) to denote g to the power X modulo p. Finally, we declare several identifiers like certDesc,rChat,oChat, and eFC. Those are used later by the agents as parameters in their exchanged messages. We explain them when they appear in the messages in the **Actions** section.

Formats: EAC has several formats, e.g., the format eac1input represents an XML structuring for seven fields: a certificate and six numeric values that describe the certificate and provide auxiliary information; we describe these six fields below when eac1input is used in the protocol message exchange. Note that in this section we only name each protocol format and specify the data types for its fields, the details of the format structure are abstracted away and left for the implementation (a Java class for each of the protocol formats). More about the formats of EAC is given in Section 5.2.2 that is dedicated to this purpose.

Knowledge: The initial knowledge for each of the participants of EAC is as follows:

PCD knows initially his name, a certificate issued by the certificate authority ca in which ca signs the public key of PCD. In EAC a certificate chain may be used instead of a single one, but in our specification we model them as a single certificate as it is a realistic abstraction. PCD also knows his public and private keys, the public key of ca, and the Diffie-Hellman base g. Additionally, PCD knows handle, didName, certDesc, rChat, oChat, auxData, noCert and null that he uses afterward as parameters in his messages. We explain them when they appear in the exchanged messages in the **Actions** section.

PICC knows initially his name, a certificate issued by the certificate authority ca in which the half key of PICC is signed. PICC also knows the public key of ca, his secret key sk(PICC), and finally the base g. Finally, PICC knows cHAT, cAR, eFC, idPICC and null; he uses these constants as parameters in his messages.

Actions: This section specifies what messages are exchanged among protocol participants as well as what data is created freshly during an EAC run.

1. PCD initiates the EAC protocol; he sends a handle handle and a didName that are both used to identify him. PCD also sends his certificate signed by ca along with a certificate description certDesc, a required and an optional Certificate Holder Authorization Template (CHAT) represented here with rCHAT and oCHAT, and some auxiliary data auxData. PCD wraps this data using the eac1input format before he sends them to PICC.

In the original specification of the EAC protocol [Ger08], PCD sends a certificate chain, in here we use a single certificate to model the chain. Finally, PCD sends this message over a *secure pseudonymous channel* that we denote by $[\texttt{PCD}]* \rightarrow *[\texttt{PICC}]$. By secure we mean that the intruder cannot get the exchanged messages over this channel, and by pseudonymous we mean that they both do not know each other. Due to the lack of tool support, we simulate this channel between any two agents A and B (that do not know each other), by: A sends first a fresh public-key to B, then B replies with a fresh symmetric key encrypted with the public-key of A. After this, they can both exchange message securely using the symmetric key to encrypt all exchanged messages among them; still without knowing each other. More about secure pseudonymous channels can be found in [MV09a]. This channel is the result of performing the PACE protocol step before EAC (we discuss PACE protocol in Section 5.3). This channel is used in all the steps of EAC.

2. After receiving the first message from PCD, PICC verifies the received certificate and extracts the public key of PCD. Then, PICC generates a random number (challenge) RC and sends it along with his identifier idPICC, a Certificate Holder Authorization Template cHAT, a Certification Authority Reference cAR, and eFC. The message has the format eac1output. Note that this message is dedicated to PCD as he is the only one able to decrypt it.
3. PCD then computes an ephemeral Diffie-Hellman half-key by generating the nonce X. Then he send his half key and the parameter noCert to PICC both formatted with eac2input. Recall that the let is an in-line macro used to improve readability.
4. PICC generates the random number Rpicc and send it to PCD formatted with eac2output.
5. Now PCD signs (with his private key) PICC's name, the received random number Rpicc and his half key $\texttt{exp}(\texttt{g},\texttt{X})$, then sends them to PICC. null is used to model the null value for any optional parameter. Note that the signed message is formatted with the format x59d and the whole message is formatted with eac2additionalinput.
6. Finally, PICC performs checks on the values of PCD half-key and Rpicc. Then he generates the random number Rmac, and computes: $\texttt{mac}(\texttt{hash}(\texttt{DHKey},\texttt{Rmac}),\texttt{PK\_PCD})$ where mac is a keyed hash, hash is a hash function, DHKey is the full Diffie-Hellman key, and PK_PCD is the ephemeral half-key of PCD that PICC received previously, i.e., $\texttt{exp}(\texttt{g},\texttt{X})$. After this computation, PICC sends the certificate issued by ca that signs his static half-key along with the mac above and the random number Rmac. Note that PCD can compute the full Diffie-Hellman key using the half-key of PICC and X that he previously

generated, thanks to the algebraic properties of the modular exponentiation exp. Note that the users of SPS do not have to specify how agents decrypt the received messages nor how they are verified (if a message is of a certain format); these details are defined in the semantics of SPS (defined in [Fut13c]) and implemented by SPS compiler.

Goals: Here we specify the goals of the EAC protocol. EAC aims at achieving mutual authentication between its two participants, PCD and PICC. The goals are explained later in the verification tools section (to check if EAC achieves them or not). We discuss the verification of EAC in Section 5.2.3.

5.2.2 EAC Formats

The EAC protocol has several formats for its XML-messages as shown in Listings 5.2- 5.6. Note that the messages are slightly simplified, i.e., Name-spaces and organizational XML attributes were left out.

```
1 EAC1INPUT(handle, did-name, certs, cert-desc, req-chat, opt-chat, aux-data,
      transact) =
2 <DIDAuthenticate>
3   handle
4   <DIDName>did-name</DIDName>
5   <AuthenticationProtocolData Protocol="urn:oid:1.3.162.15480.3.0.14">
6     {certs: c | <Certificate>c</Certificate>}
7     <CertificateDescription>cert-desc</CertificateDescription>
8     {if req-chat: <RequiredCHAT>req-chat</RequiredCHAT>}
9     {if opt-chat: <OptionalCHAT>res-chat</OptionalCHAT>}
10    {if aux-data: <AuthenticatedAuxiliaryData>aux-data</
          AuthenticatedAuxiliaryData>}
11    {if transact: <TransactionInfo>transact</TransactionInfo>}
12  </AuthenticationProtocolData>
13 </DIDAuthenticate>
```

Listing 5.2: EAC1INPUT format

```
1 EAC1OUTPUT(chat, cars, ef-ca, id-picc, challenge) =
2 <DIDAuthenticateResponse>
3   <Result>
4     <ResultMajor>http://www.bsi.bund.de/ecard/api/1.1/resultmajor#ok</
          ResultMajor>
5   </Result>
6   <AuthenticationProtocolData Protocol="urn:oid:1.3.162.15480.3.0.14">
7     {if chat: <CertificateHolderAuthorizationTemplate>chat</
          CertificateHolderAuthorizationTemplate>}
8     {cars: c | <CertificationAuthorityReference>c</
          CertificationAuthorityReference>}
9     <EFCardAccess>ef-ca</EFCardAccess>
10    <IDPICC>id-picc</IDPICC>
11    <Challenge>challenge</Challenge>
```

```
12   </AuthenticationProtocolData>
13 </DIDAuthenticateResponse>
```

Listing 5.3: EAC1OUTPUT format

```
1 EAC2INPUT(handle, did-name, certs, key, sig) =
2 <DIDAuthenticate>
3   handle
4   <DIDName>did-name</DIDName>
5   <AuthenticationProtocolData Protocol="urn:oid:1.3.162.15480.3.0.14">
6     {certs: c | <Certificate>c</Certificate>}
7     <EphemeralPublicKey>key</EphemeralPublicKey>
8     {if sig: <Signature>sig</Signature>}
9   </AuthenticationProtocolData>
10 </DIDAuthenticate>
```

Listing 5.4: EAC2INPUT format

```
1 EAC2OUTPUT(ef-cs, token, nonce, challenge) =
2 <DIDAuthenticateResponse>
3   <Result>
4     <ResultMajor>http://www.bsi.bund.de/ecard/api/1.1/resultmajor#ok</
          ResultMajor>
5   </Result>
6   <AuthenticationProtocolData Protocol="urn:oid:1.3.162.15480.3.0.14">
7     {if challenge: <Challenge>challenge</Challenge>
8      else:
9     <EFCardSecurity>ef-cs</EFCardSecurity>
10    <AuthenticationToken>token</AuthenticationToken>
11    <Nonce>nonce</Nonce>}
12  </AuthenticationProtocolData>
13 </DIDAuthenticateResponse>
```

Listing 5.5: EAC2OUTPUT format

```
1 EACADDITIONALINPUT(handle, did-name, sig) =
2 <DIDAuthenticate>
3   handle
4   <DIDName>did-name</DIDName>
5   <AuthenticationProtocolData Protocol="urn:oid:1.3.162.15480.3.0.14">
6     <Signature>sig</Signature>
7   </AuthenticationProtocolData>
8 </DIDAuthenticate>
```

Listing 5.6: EACADDITIONALINPUT format

For each of the formats, the SPS compiler provides a Java class skeleton. Here we give as an example the class for `eac1input` format, in which we have a parser, a pretty printer and other methods of `eac1input` format.

```
1 //Auto-generated by SPS
2 //Format class prototype
3  public class eac1input{
```

```
4       private byte[] field1;
5       private byte[] field2;
6       private byte[] field3;
7       private byte[] field4;
8       private byte[] field5;
9       public eac1input( byte [] input){//insert your format parser code
10      }
11      public eac1input(byte[] x1, byte[] x2, byte[] x3, byte[] x4, byte[] x5){
            //insert your format setter here
12          field1=x1;
13          field2=x2;
14          field3=x3;
15          field4=x4;
16          field5=x5;
17      }
18      public boolean verify(){
19      //insert your verifier code here
20      }
21      public byte[] get1(){
22      //insert your getter code here
23      return field1;
24      }
25      public byte[] get2(){
26      //insert your getter code here
27      return field2;
28      }
29      public byte[] get3(){
30      //insert your getter code here
31      return field3;
32      }
33      public byte[] get4(){
34      //insert your getter code here
35      return field4;
36      }
37      public byte[] get5(){
38      //insert your getter code here
39      return field5;
40      }
41      public byte[] encode(){//insert your format pretty-printer code here
42      }
43  }
```

Listing 5.7: eac1input Class

As shown Listing 5.7 we have the following methods:

- Two constructors, one that parses a byte array into an `eac1input` object (this corresponds to a deserializer), and another that constructs such an object from the given fields values.
- A `verify` method to check whether a message is a valid `eac1input` object.
- An `encode` method that returns the object in concrete syntax, also referred as a serializer or a pretty-printer.

- For each element of the object, a `get` method that returns that element.

5.2.3 Analysis Results for EAC

Here we present the formal verification aspects and results of EAC, i.e., Proverif counter examples or proofs and OFMC attacks or bounded verification.

We use EAC as our key example, i.e., we discuss in details different aspects of the analysis of this protocol. However, in the other protocols we only point out the differences if they exists, in order to avoid any redundancy.

Proverif

Using the SPS compiler, we translated the EAC specification into an Applied π code. To improve the presentation of this code, we will explain the main parts using code chunks. The first part is the *declarations* part in which we declare all identifiers and functions used in the rest of the code. Listing 5.8 shows a chunk of this part, where one can find the declaration of the following:

- A channel `ch` (Line 1) that agents use to exchange message.
- The intruder name (Line 2).
- The used mappings like `inv` (Line 3) and their only property of being `private`, i.e., agents (including the intruder) cannot use it to compose messages.
- The used functions such as the symmetric encryption function `scrypt` (Line 3) followed by its decryption property; namely that if the key is known then the message is also known. Proverif considers all functions to be public unless they are followed by `[private]` as in the mapping `inv`, thus another property of `scrypt` is that it is public (all agents including the intruder can use it to compose messages).
- The used formats, e.g., `certform` (Line 6) followed its properties (Lines 7 and 8), i.e., if one has a formatted message then he can get the fields of that formats (decompose the format). Formats are also public.
- Finally, the goals. (Line 10) is a query for Proverif to answer if the attacker can get the enclosed secret `expexpgxskpiccPICCPCD` that is declared to be private in the previous line.

```
free ch: channel.
free i:bitstring.
fun inv(bitstring):bitstring [private].
fun scrypt(bitstring,bitstring):bitstring.
 reduc forall k: bitstring, m: bitstring;  sdecrypt(k,
     scrypt(k,m))=m.
fun certform(bitstring,bitstring):bitstring.
 reduc forall a:bitstring,b:bitstring;certformget1(
     certform(a,b))=a.
 reduc forall a:bitstring,b:bitstring;certformget2(
     certform(a,b))=b.

free expexpgxskpiccPICCPCD:bitstring[private].
query attacker(expexpgxskpiccPICCPCD).
```

Listing 5.8: Declaration part of EAC in Applied π

The second part is the processes of the EAC participants, i.e., `PCD` and `PICC`. These processes contain all actions each participant performs such as sending and receiving messages, performing checks and what event he issues. The events are used later to verify goals accordingly. Listing 5.9 shows the process of `PCD`.

```
(*Agent PCD process in protocol:EAC  is *)
let processPCD(x1:bitstring,x2:bitstring,x3:bitstring,
    x4:bitstring,x5:bitstring,x6:bitstring,x7:bitstring
    ) =
new x8:bitstring;

out(secCh(x2,x1), eac1input(x3, x8));
in(secCh(x1,x2), x9:bitstring);
let x10:bitstring = eac1outputget1(x9) in
let x11:bitstring = eac1outputget2(x9) in
let x12:bitstring = eac1outputget3(x9) in
if(x11 = x1) then
new x13:bitstring;

new x14:bitstring;

out(secCh(x2,x1), eac2input(x14, exp(x7, x13)));
in(secCh(x1,x2), x15:bitstring);
let x16:bitstring = eac2outputget1(x15) in
```

```
18 out(secCh(x2,x11), eac2additionalinput(sign(x6, x59d(
       x11, x16, exp(x7, x13)))));
19 in(secCh(x11,x2), x17:bitstring);
20 let x18:bitstring = eac22outputget1(x17) in
21 let x19:bitstring = eac22outputget2(x17) in
22 let x20:bitstring = eac22outputget3(x17) in
23 let x21:bitstring = open(x5, x18) in
24 let x22:bitstring = certformget1(x21) in
25 let x23:bitstring = certformget2(x21) in
26 if(x22 = x11) then
27 if(x23 = exp(x7, sk(x11))) then
28 if(x19 = mac(hash((exp(exp(x7, x13), sk(x11)), x20)),(
       exp(x7, x13)))) then
29 if(x1<> i) then
30 out(ch, scrypt(hash((sk(x1), exp(x7, x13), x20)),
       hashskpiccexpgxrmacPICCPCD));out(ch, scrypt(hash((
       sk(x1), exp(x7, x13), x20)),
       hashskpiccexpgxrmacPICCPCD));0.
31 (* ------------------------- *)
32          (*EOP*)
```

Listing 5.9: Process of PCD in Applied π

The third part is the "global" process that instantiates the above processes with their initial knowledges and allows the intruder to instantiate some of these process to model the dishonest agents behavior. One important feature here is that our generated code verifies EAC for unbounded number of session and agents. Listing 5.9 shows part of PCD.

```
1 process
2 new ca:bitstring;
3
4 !new x:bitstring;
5 out(ch, x)|
6 !in(ch,(picc:bitstring));
7 processPCD(picc, x, sign(inv(pk(ca)), certform(x, pk(x
      ))), pk(x), pk(ca), inv(pk(x)), g) |
8 out(ch,(picc, i, sign(inv(pk(ca)), certform(i, pk(i)))
      , pk(i), pk(ca), inv(pk(i)), g))|
9 !in(ch,(pcd:bitstring));
```

```
10 processPICC(pcd, x, sign(inv(pk(ca)), certform(x, exp(
       g, sk(x)))), pk(ca), sk(x), g) |
11 out(ch,(pcd, i, sign(inv(pk(ca)), certform(i, exp(g,
       sk(i)))), pk(ca), sk(i), g))
```

Listing 5.10: The "Global" process of EAC in Applied π

The summary of Proverif analysis of EAC is shown in Table 5.1

Table 5.1: Proverif Analysis Summary for EAC

Goal of EAC	Proverif result
Secrecy Goal `DHKey secret of PICC, PCD`	Proof found
Authentication Goals `PCD authenticates PICC on DHKey`	Proof found
`PICC authenticates PCD on DHKey`	Proof found

OFMC

SPS compiler also generates AVISPA IF specifications that can be checked with the OFMC. The summary of the OFMC results for EAC are shown in Table 5.2.

Table 5.2: OFMC Analysis Summary for EAC

Goal of EAC	OFMC result
Secrecy Goal `DHKey secret of PICC, PCD`	No attack found
Authentication Goals `PCD authenticates PICC on DHKey`	No attack found
`PICC authenticates PCD on DHKey`	No attack found

As shown in the Tables 5.1 and 5.2, both of the back-end tools agreed that EAC has no attack.

5.3 PACE

The Password Authenticated Connection Establishment (PACE) [Fed12] protocol establishes a strong session key for secure communication based on a shared, weak secret. PACE is used by the German identity card (i.e. `PICC`) to (1) protect the communication over the contact-less interface between the card and the terminal (i.e. `PCD`) and (2) authenticate the legitimate card holder using a PIN. It was invented by the German Federal Office for Information Security as a replacement for the insecure Basic Access Control (BAC).

In brief, PACE works as follows: (1) The card user `User` enters the PIN into the card reader `PCD`. (2) `PCD` requests the card `PICC` to send him a freshly generated random number encrypted with the PIN. (3) `PICC` creates a random number, encrypts it with the shared secret (i.e. PIN), and sends it to the `PCD`. (4) `PCD` decrypts the received message to get the random number, creates an ephemeral key pair, and sends the public key to the `PICC`. (5) `PICC` also generates an ephemeral key pair and responses with the public key. (6) Both perform an elliptic curve Diffie-Hellman key agreement and compute a common shared secret. (7) Both compute a new Diffie-Hellman generator based on the common shared secret and the random number, and then perform an additional Diffie-Hellman key agreement using the new generator. Finally, (8) both derive session keys for encryption and establish a secure channel between them. The channel between `User` and `PCD` is a pseudonymous secure channel, i.e., they can exchange messages securely without knowing each other (relying on a pseudonym instead of their real names). We model this channel by: first, `User` generates a public/private key pair (we only model the generation of the public part and the private part is implicit, i.e., if an agent creates a public key, then the private key is added to his knowledge implicitly), then he sends his key and the PIN to `PCD` encrypted with the public key of `PCD`. We also add a step in the end of the protocol to enable the check authentication between `PICC` and `User`.

Please note that the `PICC` and `PCD` exchange information about key sizes, encryption and hash algorithms, Diffie-Hellman generators, and so forth beforehand; and therefore we include this information in their initial knowledges as shown shortly.

5.3.1 PACE in SPS

In this section we present the PACE protocol in SPS (cf. Listing 5.11) and describe it in detail.

```
1 Protocol: PACE
```

```
Types:
 Agent  PCD, PICC, User;
 Number g, three, oid, secretid, null;

Formats:
 mseSetATRequest(Msg, Msg);
 mseSetATResponse(Msg);
 encryptedNonceRequest(Number);
 encryptedNonceResponse(Number);
 mapNonceRequest(PublicKey);
 mapNonceResponse(PublicKey);
 performKeyAgreementRequest(PublicKey);
 performKeyAgreementResponse(PublicKey);
 mutualAuthenticationRequest(Number);
 mutualAuthenticationResponse(Number);

Knowledge:
 PICC: PICC, User, g, three, shk(User,PICC), oid, secretid, null;
 PCD:  PCD, g, three, oid, secretid, pk(PCD), inv(pk(PCD)), null;
 User: User, PICC, PCD, shk(User,PICC), pk(PCD), null;
 where PCD != i;

Actions:
 User: PublicKey PkA
 User -> PCD: crypt(pk(PCD), (shk(User, PICC), PkA))

 PCD -> PICC: mseSetATRequest(oid, secretid)

 PICC -> PCD: mseSetATResponse(null)

 PCD -> PICC: encryptedNonceRequest(null)

 PICC: Number R
 PICC -> PCD: encryptedNonceResponse(scrypt(shk(User,PICC), R))

 PCD: Number X1
 PCD -> PICC: mapNonceRequest(mult(X1,g))

 PICC: Number Y1
 PICC -> PCD: mapNonceResponse(mult(Y1,g))

 PCD: Number X2

 let PCD_K1 = mult(mult(Y1,g), X1)
 let PCD_g2 = add(mult(R,g), PCD_K1)
 PCD -> PICC: performKeyAgreementRequest(mult(X2, PCD_g2))

 PICC: Number Y2

 let PICC_K1 = mult(mult(X1,g), Y1)
 let PICC_g2 = add(mult(R,g), PICC_K1)
 PICC -> PCD: performKeyAgreementResponse(mult(Y2, PICC_g2))

 let PCD_K2 = mult(X2, mult(Y2, PICC_g2))
 PCD -> PICC: mutualAuthenticationRequest(mac(hash(PCD_K2, three), mult(Y2,
      PICC_g2)))

 let PICC_K2 = mult(mult(X2, PCD_g2), Y2)
```

```
59 let Key = mult(X2, PCD_g2)
60 PICC: Number NPICC
61 PICC-> PCD: mutualAuthenticationResponse(mac(hash(PICC_K2,three),mult(X2,
       PCD_g2))), scrypt(Key ,NPICC)
62
63 PCD-> User: crypt(PkA, (shk(User, PICC), NPICC))
64
65 Goals:
66 Key secret of PCD, PICC
67 NPICC secret of PICC, User
68 User authenticates PICC on NPICC
```

Listing 5.11: PACE in SPS

Protocol: This section specifies the name of the protocol, PACE.

Types: This section specifies the three agents that are involved in the PACE protocol: (1) the `PCD` representing the terminal, (2) the `PICC` representing the eID card, and (3) the user `User`. Furthermore, this section specifies the numbers `g`, `three`, `oid`, `secretid`, and `null`. `g` represents a generator for the Diffie-Hellman key exchange. `three` represents simply the number three that is used as a constant in the key derivation. `oid` is the object identifier that specifies the algorithms that should be used in the protocol. `secretid` tells what secret should be used, i.e., PIN in our case, but it could be PUK or CAN as well. Finally, *null* represents the empty value for optional parameters.

Formats: In this section we defines various formats for the PACE protocol. The PACE formats come in request/response pairs, e.g., we have `mseSetATRequest` and `mseSetATResonce` formats; we refer to this pair of format by the `mseSetAT`, that in fact abbreviates Manage Security Environment Set Authentication Template. `mseSetAT` formats are used to initialize the PACE protocol. The `encryptedNonce` formats represent the exchange of the encrypted random number. The `mapNonce` message formats are used for the first Diffie-Hellman key exchange, and the `performKeyAgreement` formats for the second key exchange. The formats of `mutualAuthentication` are used to convey the authentication token.

Knowledge: The initial knowledge of the participants is as follows:

`PICC` knows his name, the user name `User`, and the constants `g`, `three`, `oid`, and `secretid`. He also knows `shk(User, PICC)` a shared key between him and the user `User`; this shared key models the PIN.

`PCD` knows his name, and the constants `g`, `three`, `oid`, and `secretid`. He also has a public/private key pair represented by `pk(PCD)` and `inv(pk(PCD))`.

`User` knows his name, the names of `PCD` and `PICC`. He also knows the public key of `PCD` and the PIN, i.e., $\texttt{shk}(\texttt{User}, \texttt{PICC})$. The reason behind the user knowing the public key of `PCD` is to model the secure pseudonym channel between both of them as mentioned earlier.

Actions: We explain this section step-by-step.

1. The user `User` sends the PIN ($\texttt{shk}(\texttt{User}, \texttt{PICC})$) to `PCD`. For lack of tool support and to model that `User` and `PCD` are not known to each other, but the user can send the PIN securely to `PCD`; we model the channel between them by: first, the user generates a public/private key pair (we only model the generating of the public part and the private part is implicit), then he sends his key and the PIN to the terminal `PCD` encrypted with the public key of `PCD`.
2. `PCD` derives the PIN ($\texttt{shk}(\texttt{User}, \texttt{PICC})$) from the received message, then sends the `mseSetAT` message to the `PICC`. The `oid` is an object identifier that specifies the PACE version and cryptographic algorithms that should be used in the protocol run. The `secretid` specifies the type of the secret (PIN, PUK, or CAN), in our case, we assume it is always the PIN.
3. Subsequently, the `PCD` requests an encrypted random number from the `PICC`.
4. The `PICC` generates the random number `R`, encrypts it using the shared key $\texttt{shk}(\texttt{User}, \texttt{PICC})$, and sends it to the `PCD`.
5. The `PCD` decrypts the message and retrieves `R`. It generates a secret `X1`, computes $\texttt{mult}(\texttt{X1}, \texttt{g})$, and sends it to `PICC`. Please note that is step is the first part of the Diffie-Hellman key agreement and that `mult` represents the modular multiplication.
6. In response, `PICC` generates the secret `Y1` and sends $\texttt{mult}(\texttt{Y1}, \texttt{g})$ back.
7. The `PCD` finishes the first key agreement by computing the first Diffie-Hellman shared secret `PCD_K1`. Now, `PCD` starts the second key agreement. He computes the new generator `PCD_g2`, generates `X2`, computes $\texttt{mult}(\texttt{X2}, \texttt{g2})$, and sends it to `PICC`.
8. The `PICC` acts similar. It computes `PICC_K1`, generates `Y2`, computes $\texttt{mult}(\texttt{Y2}, \texttt{g2})$, and sends it to `PCD`.
9. Now both participants can calculate a shared authentication token. First we show how `PCD` does that (in the next step we show it for `PICC`):
   (a) `PCD` calculates a common shared key $\texttt{mult}(\texttt{mult}(\texttt{g}, \texttt{X2}), \texttt{Y2})$.
   (b) Then he derives a MAC key using the hash function and the constant three

$\texttt{hash}(\texttt{mult}(\texttt{X2}, \texttt{mult}(\texttt{Y2}, \texttt{add}(\texttt{mult}(\texttt{R}, \texttt{g}), \texttt{mult}(\texttt{mult}(\texttt{X1}, \texttt{g}), \texttt{Y1})))), \texttt{three})$

10. Finally, PICC calculates the shared authentication token in a similar way to what PCD did in the last step, i.e.:
    (a) PICC calculates a common shared key $\texttt{mult}(\texttt{mult}(\texttt{g}, \texttt{X2}), \texttt{Y2})$.
    (b) Then he derives a MAC key using the hash function and the constant three

    $\texttt{hash}(\texttt{mult}(\texttt{Y2}, \texttt{mult}(\texttt{X2}, \texttt{add}(\texttt{mult}(\texttt{R}, \texttt{g}), \texttt{mult}(\texttt{mult}(\texttt{Y1}, \texttt{g}), \texttt{X1})))), \texttt{three})$

Goals: The goal of PACE is to establish a secure channel between the PICC and the PCD. More about PACE goals is found in Section 5.3.3.

5.3.2 PACE Formats

The formats that PACE uses to structure its messages are basically Application Protocol Data Units (APDU). Listing 5.12 specifies the APDUs in PACE using the following notation; which is based on the notation of [MK14]) and extended with $\mathsf{tlv}(\cdot, \cdot)$ to represent tag-length-value as we explain shortly.

- $\mathsf{byte}(n)$ denotes one-byte constant n, e.g., $\mathsf{byte}(128)$ means the constant 128 represented in a byte.
- $\cdot$ denotes the concatenation operator, e.g., $\mathsf{byte}(3) \cdot oid$ means the concatenation of the constant 3 with the string oid.
- $\mathsf{off}_n(data)$ means an offset (with a fixed length of n bytes) that tells how many bytes the following $data$ is supposed to be, e.g., $\mathsf{off}_3(secretID)$ means that the number presented in the first 3 bytes tells what is the length of the following $secretID$.
- $\mathsf{tlv}(\mathsf{tag}, \mathsf{value})$ represents a tag-length-value encoding, e.g., $\mathsf{tlv}(\mathsf{byte}(124), \mathsf{X1})$ says that $\mathsf{X1}$ is of variable length and prefixed with the constant 124.

5.3.3 Analysis Results for PACE

Proverif

We use the SPS compiler to generate Applied π code from PACE specification. Unfortunately, we cannot use Proverif to verify PACE because the algebraic

$$\texttt{mseSetATRequest}(oid, secretID) = \mathsf{byte}(0) \cdot \mathsf{byte}(32) \cdot \mathsf{byte}(193) \cdot \mathsf{byte}(164) \cdot \mathsf{tlv}(\mathsf{byte}(128), oid) \cdot \mathsf{tlv}(\mathsf{byte}(131), secretID)$$

$$\texttt{mseSetATResponse}() = \mathsf{byte}(140) \cdot \mathsf{byte}(0)$$

$$\texttt{encryptedNonceRequest}() = \mathsf{byte}(16) \cdot \mathsf{byte}(134) \cdot \mathsf{byte}(0) \cdot \mathsf{byte}(0) \cdot \mathsf{byte}(2) \cdot \mathsf{byte}(124) \cdot \mathsf{byte}(0) \cdot \mathsf{byte}(0)$$

$$\texttt{encryptedNonceResponse}(random) = random \cdot \mathsf{byte}(140) \cdot \mathsf{byte}(0)$$

$$\texttt{mapNonceRequest}(X1) = \mathsf{byte}(16) \cdot \mathsf{byte}(134) \cdot \mathsf{byte}(0) \cdot \mathsf{byte}(0) \cdot \mathsf{tlv}(\mathsf{byte}(129), \mathsf{tlv}(\mathsf{byte}(124), \mathsf{X1})) \cdot \mathsf{byte}(0)$$

$$\texttt{mapNonceResponse}(Y1) = \mathsf{tlv}(\mathsf{byte}(130), \mathsf{tlv}(\mathsf{byte}(124), \mathsf{Y1})) \cdot \mathsf{byte}(140) \cdot \mathsf{byte}(0)$$

$$\texttt{performKeyAgreementRequest}(X2) = \mathsf{byte}(16) \cdot \mathsf{byte}(134) \cdot \mathsf{byte}(0) \cdot \mathsf{byte}(0) \cdot \mathsf{tlv}(\mathsf{byte}(131), \mathsf{tlv}(\mathsf{byte}(124), \mathsf{X1})) \cdot \mathsf{byte}(0)$$

$$\texttt{performKeyAgreementResponse}(Y2) = \mathsf{byte}(16) \cdot \mathsf{byte}(134) \cdot \mathsf{byte}(0) \cdot \mathsf{byte}(0) \cdot \mathsf{tlv}(\mathsf{byte}(132), \mathsf{tlv}(\mathsf{byte}(124), \mathsf{Y1})) \cdot \mathsf{byte}(0)$$

$$\texttt{mutualAuthenticationRequest}(token) = \mathsf{byte}(16) \cdot \mathsf{byte}(134) \cdot \mathsf{byte}(0) \cdot \mathsf{byte}(0) \cdot \mathsf{tlv}(\mathsf{byte}(133), \mathsf{tlv}(\mathsf{byte}(134), \mathsf{token}))$$

$$\texttt{mutualAuthenticationResponse}(token) = \mathsf{byte}(16) \cdot \mathsf{byte}(134) \cdot \mathsf{byte}(0) \cdot \mathsf{byte}(0) \cdot \mathsf{tlv}(\mathsf{byte}(134), \mathsf{tlv}(\mathsf{byte}(134), \mathsf{token}))$$

Listing 5.12: PACE Formats

properties of `mult` cannot be handled by Proverif. More precisely, as `mult` properties that we need here are an AC theory, this implies the generating of an infinite number of rewrite rules in Proverif and thus causing non-termination. In a nutshell, Proverif cannot handle associative operators in general [BS11]. This is an example of the limitations that we may face that are "inherited" from the tools and invites for further research for improving existing tools especially in the handling of algebraic properties.

OFMC

The results of checking PACE with OFMC is summarized in Table 5.3.

Table 5.3: OFMC Analysis Summary for PACE

Goal of PACE	OFMC result
Secrecy	
`Key secret of PCD, PICC`	No attack found for 2 sessions
`NPICC secret of PICC, User`	No attack found for 2 sessions
Authentication	
`User authenticates PICC on NPICC`	No attack found for 2 sessions

5.4 TLS

The Transport Layer Security (TLS) protocol [DR08, Eas11] is a widely known protocol that establishes a secure channel between a client and a server. TLS is widely used in many applications including eID systems; several European eID systems use it for authentication, e.g., the Italian and Swiss eID systems [Fut13a].

5.4.1 TLS in SPS

Here we present TLS specification in SPS based on [MK14]

```
1  Protocol: TLS
2  Types:
3   Agent A,B,ca;
4   Number eps,cipher,compr,t20,t22,t23;
5  Formats:
6   clientHello(Msg, Msg, Msg, Msg, Msg);
7   serverHello(Msg, Msg, Msg, Msg, Msg);
8   serverCert(Msg);
9   serverHelloDone(Msg);
10  clientKeyExchage(Msg);
11  finished(Msg);
12  pmsForm(Msg);
13  masterForm(Msg,Msg);
14  clientFinished(Msg,Msg,Msg);
15  serverFinished(Msg,Msg,Msg);
16  keyBlock(Msg, Msg);
17  changeCipher(Msg);
18     ####
19  record(Number, Msg);
20  x509(Agent, PublicKey);
21
22 Knowledge:
23  A: A,B,pk(ca),eps,cipher,compr, shk(A,B), t20,t22,t23;
24  B: B, pk(B),pk(ca),inv(pk(B)), sign(inv(pk(ca)),x509(B,pk(B))),eps,cipher,
        compr, shk(A,B), t20,t22,t23;
25 Actions:
```

```
# A generates Ra and T1
 A: Number Ra, T1
 let AM1= record(t22, clientHello(T1,Ra,eps,cipher,compr))
 A->B: AM1
#B generates Rb, Id T2 and
 B: Number Rb, T2, Id
 let BM1=record(t22, serverHello(T2,Rb,Id,cipher,compr)),
        record(t22, serverCert(sign(inv(pk(ca)),x509(B,pk(B)))),
        record(t22, serverHelloDone(eps))
 B->A: BM1
#A checks the certificate for a TTP ca
#A extracts the public key of B
#A generates the Pre-Mster Secret PMS.
#A computes MS=PRF(mster-form(PMS;RA + RB))
 let MS = prf(masterForm(PMS,add(Ra,Rb)))
 A: Number PMS
 let AM2= record(t22, clientKeyExchage(crypt(pk(B), pmsForm(PMS)))),
        record(t20, changeCipher(eps)),
        record(t22, finished(prf(clientFinished(MS,add(Ra,Rb),hash(AM1, BM1))
            )))
 A->B: AM2
 B->A: record(t20, changeCipher(eps)), record(t22, finished(prf(
     serverFinished(MS,add(Ra,Rb),hash(AM1, BM1, AM2)))))
#A computes the key clntK=extractCK(key block(MS;RA + RB))
#A and B exchange payload messages as follows:
 A: Number PAYLOADA
 A->B: record(t23, scrypt(extCK(keyBlock(MS, add(Ra,Rb))), shk(A,B)))
#B computes srvrK=extractSK(key block(MS;RA + RB))
 B: Number PAYLOADB
 B->A: record(t23, scrypt(extSK(keyBlock(MS, add(Ra,Rb))), PAYLOADB))

Goals:
 MS secret of A,B
 B authenticates A on MS
```

Listing 5.13: TLS in SPS

Now we explain the TLS specification given above by section.

Protocol: In this section we give the name of the protocol, TLS.

Types: In TLS, we have three agents: (1) `A` represents a client, (2) `B` represents a server, and (3) `ca` represents a trusted certificate authority that issued a certificate for the server `B`. We also declare the cipher suite `cipher`, the compression algorithm `compr`, the empty string `eps`, and the tags `t20`, `t22` and `t23`. We later explain how agents use them.

Formats: TLS has several formats that structure its messages. For instance, the format `helloClient` has five fields of type `Msg`: a time-stamp, a freshly generated number, a session-id, cipher suites and compression methods. Later we provide more details about TLS formats.

Knowledge: The initial knowledge for each of the participants of TLS is as follows:

- `A` knows initially his name, the name of `B`, and the public key of the trusted third party `ca`. She also knows the cipher suite `cipher`, the compression algorithm `compr` and the empty string `eps`. Finally she knows the tags `t20, t22` and `t23`.
- `B` knows initially his name, the name of `A`, his private and public keys and the public key of the trusted third party `ca` (certificate authority). He also has a certificate issued for him by `ca` and he knows the cipher suite `cipher`, the compression algorithm `compr` and the empty string `eps`. Finally he knows the tags `t20, t22` and `t23`.

Actions: In this section, we specify the exchange messages between protocol participants as well as the freshly generated data during a TLS execution.

1. The client `A` initiates the protocol; she generates a fresh random number `Ra` and a time-stamp `T1` and sends them to the server `B` along with the empty-string `eps`, her preferences for encryption `cipher`, and her preferences for compression algorithms `compr`. She formats this data with the format `clientHello`, and envelopes this message with the format `record`. The tag `t22` denotes that this message belongs to the handshake sub-protocol. We refer to the whole message by `AM1` for later reference.
2. The server `B`, in response, generates the fresh random number Rb, the time-stamp `T2`, and the session-id `Id`. Then with the received `cipher`, and `compr` he formats them using the `serverHello` format. He also sends his certificate that has the `x509` format and enveloped with `serverCert` format. (This certificate is issued by the trusted third party `ca`.) Each of these messages (the `serverHello`, `serverCert`, and the `serverHelloDone`) is enveloped with the format `record` then sent to the client `A`. The tag `t22` denotes that this message belongs to the TLS-handshake sub-protocol. We refer to this whole message by `BM1` for later reference as well.
3. Now, `A` checks the certificate issued by `ca` and extracts the public key of `B`. Then he generates the Pre-Master Secret `PMS` and computes $\texttt{MS} = \texttt{prf}(\texttt{masterForm}(\texttt{PMS}, \texttt{add}(\texttt{Ra}, \texttt{Rb})))$ using the pseudo-random function `prf`. He encrypts `MS` with the public key of `B` and formats it with `clientKeyExchange`.

   Then each of: (1) the previous message, (2) the change-cipher message: $\texttt{changeCipher}(\texttt{eps})$, and (3) the client-finished message that is: $\texttt{finished}(\texttt{prf}(\texttt{clientFinished}(\texttt{MS}, \texttt{add}(\texttt{Ra}, \texttt{Rb}), \texttt{hash}(\texttt{AM1}, \texttt{BM1}))))$; is enveloped with `record` and sent to `B`. Note that `A` included the hashes

of the previous messages AM1 and BM1. Also note that the finished format is nesting another format, namely clientFininshed, and that the tag of the second message (2) is t20 denoting that it belongs to the TLS sub-protocol: change-cipher specification.

4. B computes the pseudo-random function of the server-finished that is: serverFinished(MS, add(Ra, Rb)) and the hash of all previous messages, i.e., AM1, BM1 and AM2. He wraps the computed value with the format finished and sends it to A as his finished message.
5. A computes the key extCK(keyBlock(MS, add(Ra, Rb))) using the client extract-key function extCK and uses it to encrypt the payload message PAYLOADA, then sends the encrypted message to B. Here (and in the next) the record message is tagged with t23 to denote that it belongs to the TLS sub-protocol: application data.
6. B also computes the key extSK(keyBlock(MS, add(Ra, Rb))) using the server extract-key function extCK and uses it to encrypt the payload message PAYLOADB, then sends the encrypted message to A. Note that both extCK and extSK are publicly known functions, so both the client and the server can compute the keys of the encrypted messages sent to them.

Goals: TLS aims at establishing a secure channel between A and B so they can securely exchange payloads. The goals are explained later in the analysis results of TLS (Section 5.4.4).

5.4.2 TLS Formats

The formats used in the TLS protocol as specified in [MK14] are shown in Table 5.4. In this table we use the same notation that we used in PACE.

For example, consider the very first format of Table 5.4, clientHello that has five fields: a time-stamp, a freshly generated random number, a session-id, cipher suites and compression methods. This format is structured as follows:

- Three bytes with the values 1, 3 and 3, where the latter two bytes indicate the TLS version number, namely 1.2.
- A fixed-length field for the time-stamp.
- A fixed-length field for the random number.
- One-byte offset that tells the length of the following field reserved for the session-id.

- Two-byte offset that tells the length of the following field reserved for the cipher suites.

- One-byte offset that tells the length of the following field reserved for the compression methods.

Table 5.4: TLS formats

$\texttt{clientHello}(time, random, session\_id, cipher\_suites, comp\_methods)$
$= \text{HANDSHAKE}(\mathsf{byte}(1), \mathsf{byte}(3) \cdot \mathsf{byte}(3) \cdot time \cdot random \cdot \mathsf{off}_1(session\_id) \cdot$
$\mathsf{off}_2(cipher\_suites) \cdot \mathsf{off}_1(comp\_methods))$

$\texttt{serverHello}(time, random, session\_id, chosen\_cipher, chosen\_comp)$
$= \text{HANDSHAKE}(\mathsf{byte}(2), \mathsf{byte}(3) \cdot \mathsf{byte}(3) \cdot time \cdot random \cdot \mathsf{off}_1(session\_id) \cdot$
$chosen\_cipher \cdot chosen\_comp)$

$\texttt{serverCert}(certificate\_tls\_vec)$
$= \text{HANDSHAKE}(\mathsf{byte}(11), \mathsf{off}_3(certificate\_tls\_vec))$

$\texttt{serverHelloDone}() = \text{HANDSHAKE}(\mathsf{byte}(14), \epsilon)$

$\texttt{clientKeyExchange}(EncrPreMasterSecret)$
$= \text{HANDSHAKE}(\mathsf{byte}(16), EncrPreMasterSecret)$

$\texttt{finished}(encr\_finished) = \text{HANDSHAKE}(\mathsf{byte}(20), encr\_finished)$

$\texttt{pmsForm}(secret) = \mathsf{byte}(3) \cdot \mathsf{byte}(3) \cdot secret$

$\texttt{masterForm}(PMS, R) = PMS \cdot \texttt{"master secret"} \cdot R$

$\texttt{clientFinished}(MS, R, H) = MS \cdot \texttt{"client finished"} \cdot R \cdot H$

$\texttt{serverFinished}(MS, R, H) = MS \cdot \texttt{"server finished"} \cdot R \cdot H$

$\texttt{keyBlock}(MS, R) = MS \cdot \texttt{"key expansion"} \cdot R$

$\texttt{changeCipher}() = \mathsf{byte}(1)$

$\texttt{certRequest}(cert\_type, supp\_alg, cert\_auths)$
$= \text{HANDSHAKE}(\mathsf{byte}(13), \mathsf{off}_1(cert\_type) \cdot$

$\texttt{clientCert}(certificate\_tls\_vec)$
$= \text{HANDSHAKE}(\mathsf{byte}(11), \mathsf{off}_3(certificate\_tls\_vec))$

$\texttt{certVerify}(signed\_handshake) = \text{HANDSHAKE}(\mathsf{byte}(15), signed\_handshake)$

$\text{RECORD}(sub, data) = sub \cdot byte(3) \cdot byte(3) \cdot \mathsf{off}_2(data) \cdot data$

5.4.3 TLS with client authentication

We also consider another version of TLS with client authentication. For brevity, we only point out the main differences with the previously detailed version of TLS. In this version, the client `A` has a certificate issued by the trusted third party `ca` that he sends to the server `B` in the third step of the protocol in response to a certificate request from the server in the step before. One further difference between the two versions of TLS is that the latter one (TLS with client authentication) achieves an extra goal, namely that the client `A` can be authenticated to the server `B`. It is well known that this goal cannot be achieved in TLS without client certificate.

```
Protocol: TLS-CA #with client certificate
Types:
 Agent A,B,ca;
 Number eps,cipher,compr,t20,t22,t23;
Formats:
 clientHello(Msg, Msg, Msg, Msg, Msg);
 serverHello(Msg, Msg, Msg, Msg, Msg);
 serverCert(Msg);
 serverHelloDone(Msg);
 clientKeyExchage(Msg);
 finished(Msg);
 pmsForm(Msg);
 masterForm(Msg,Msg);
 clientFinished(Msg,Msg,Msg);
 serverFinished(Msg,Msg,Msg);
 keyBlock(Msg, Msg);
 changeCipher(Msg);
 certRequest(Msg, Msg, Msg);
 clientCert(Msg);
 certVerify(Msg)
    ###
 record(Number, Msg);
 x509(Agent, PublicKey);

Knowledge:
 A: A,B,pk(A),pk(ca),inv(pk(A)), sign(inv(pk(ca)),x509(A,pk(A))),eps,cipher,
      compr, t20,t22,t23;
 B: B, pk(B),pk(ca),inv(pk(B)), sign(inv(pk(ca)),x509(B,pk(B))),eps,cipher,
      compr,t20,t22,t23, certType, suppAlg, certAuth;

Actions:
# A generates Ra and T1
 A: Number Ra, T1
 let AM1= record(t22, clientHello(T1,Ra,eps,cipher,compr))
 A->B: AM1
#B generates RB T2 and Id
 B: Number Rb, Id, T2
 let BM1= record(t22, serverHello(T2,Rb,Id,cipher,compr)),
        record(t22, serverCert(sign(inv(pk(ca)),x509(B,pk(B)))),
        record(t22, certRequest(certType,suppAlg,certAuth)),
        record(serverHelloDone(eps))
 B->A: BM1
```

```
#A checks the certificate for a TTP ca
#A extracts the public key of B
#A generates the Pre-Mster Secret PMS.
#A computes MS=PRF(mster-form(PMS;RA + RB))
# Here in certVerify(eps), eps replaces signed_hanshake
 let MS = prf(masterForm(PMS,add(Ra,Rb)))
 A: Number PMS
 let AM2= record(t22, clientCert(sign(inv(pk(ca)),x509(A,pk(A)))),
        record(t22, clientKeyExchage(crypt(pk(B), pmsForm(PMS)))),
        record(t22, certVerify(eps)),
        record(t20, changeCipher(eps)),
        record(t22, finished(prf(clientFinished(MS,add(Ra,Rb),hash(AM1, BM1))
            )))
 A->B: AM2

 B->A: record(t20, changeCipher(eps)), record(t22, finished(prf(
      serverFinished(MS,add(Ra,Rb),hash(AM1, BM1, AM2)))))
#A computes the key clntK=extractCK(key block(MS;RA + RB))
#A and B exchange payload messages as follows:
 A: Number PAYLOADA
 A->B: record(t23, scrypt(extCK(keyBlock(MS, add(Ra,Rb))), PAYLOADA))
#B computes srvrK=extractSK(key block(MS;RA + RB))
 B: Number PAYLOADB
 B->A: record(t23, scrypt(extSK(keyBlock(MS, add(Ra,Rb))), PAYLOADB))
Goals:
 MS secret of A,B
 A authenticates B on MS
 B authenticates A on MS
```

Listing 5.14: TLS with Client authentication in SPS

5.4.4 Analysis Results for TLS

As mentioned before, we provide in this section the formal verification results that we obtained from running the back-end verification tools on our auto-generated code from the SPS compiler.

Proverif

The results obtained from Proverif are summarized in Table 5.5.

Table 5.5: Proverif Analysis Summary for TLS

Goal of TLS	Proverif result no client auth.	Proverif result with client auth.
Secrecy `MS secret of A,B`	Proof found	Proof found

Authentication
`A authenticates B on MS`
`B authenticates A on MS`
	—
	Proof found
		Proof found
		Proof found

OFMC

The results of checking the two versions of TLS using OFMC is summarized in Table 5.3.

Table 5.6: OFMC Analysis Summary for TLS

Goal of TLS	OFMC result no client auth. (for 2 sessions)	OFMC result with client auth. (for 2 sessions)
Secrecy `MS secret of A,B`	No attack found	No attack found

Authentication
`A authenticates B on MS`
`B authenticates A on MS`
	—
	No attack found
		No attack found
		No attack found

5.5 ISO/IEC 9798-4

The ISO/IEC 9798 Standard specifies a family of entity authentication protocols. It comprises six main documents. The first document (Part 1 [Int10a]) is general and describes the main notions which are common for the other parts. The protocols are grouped into five parts. Part 2 [Int08] describes six protocols using symmetric encryption, Part 3 [Int10b] describes seven protocols using digital signatures, Part 4 [Int99] describes four protocols using cryptographic check functions such as MACs, Part 5 [Int09] considers protocols using zero knowledge and Part 6 [Int10c] describes eight entity authentication mechanisms based on manual data transfer between authenticating devices techniques.

5.5.1 ISO/IEC 9798-4 in SPS

Part 4 [Int99] describes two mechanisms that are concerned with the authentication of a single entity (unilateral authentication), while the remaining are mechanisms for mutual authentication of two entities. The mechanisms specified in this part of ISO/IEC 9798 use time variant parameters such as time stamps, sequence numbers, or random numbers, to prevent valid authentication information from being accepted at a later time or more than once. If a time stamp or sequence number is used, one pass is needed for unilateral authentication, while two passes are needed to achieve mutual authentication. If a challenge and response method employing random numbers is used, two passes are needed for unilateral authentication, while three passes are required to achieve mutual authentication.

In this section we consider only the two unilateral mechanism (the first and the second protocols described in the document[1]) denoted ISO9798-4.1 and ISO9798-4.2. In these authentication mechanisms the entities A and B shall share a common secret authentication key or two unidirectional secret keys prior to the commencement of any particular run of the authentication mechanisms.

The use of the text fields specified in the following mechanisms is outside the scope of this part of ISO/IEC 9798 (they may be even empty), and will depend upon the specific application. A text field may only be included in the input to the cryptographic check function if the verifier can determine it independently, e.g., if it is known in advance, sent in clear or can be derived from one or both of those sources.

Here we present the first protocol of the ISO 9797-4 specification in SPS based on [Int99]

```
1  Protocol: ISO979841
2
3  Types:
4      Agent A,B;
5      Function hash;
6  Formats:
7      tokenAB979841 (Number,Number,Msg);
8      # Agent is optional in the standard
9      fkab(Number,Agent,Number);
10     farg(SymmetricKey,Msg);
11     fm1(Msg,Agent,Number);
12
13 Knowledge:
14     A: A,B,shk(A,B);
15     B: B,A,shk(A,B);
```

[1] Since the protocols do not have a name, we identify them by the order of appearance in the document

```
16 Actions:
17     A: Number NA, Text1, Text2
18     A->B: fm1(tokenAB979841(NA,Text2,hash(farg(shk(A,B),fkab(NA,B,Text1)))),B
           ,Text1)
19
20 Goals:
21     B authenticates A on Text1
22
23 Private:
24     shk(A,B)
```

Listing 5.15: 9798-4.1 in SPS

Now we explain the protocol specification given above by section.

Protocol: In this section we give the name of the protocol, ISOCCFOnePassUnilateralAuth.

Types: Here we declare the protocol identifiers and annotate them with types. Those identifiers include the agents of the protocol: a claimant `A` and a verifier `B`. We also declare a function `hash` which is used as a cryptographic check function. As defined in ISO/IEC 9798-1, `hash(K,X)` denotes the cryptographic check value computed by applying the cryptographic check function `hash` to the data `X` using the key `K`.

Formats: Formats are detailed in 5.5.2.

Knowledge: In this section we give the initial knowledge for each of the protocol *participants*; a participant is an agent involved in the message exchanging in a protocol. The participants use the terms/messages in their initial knowledge to compose the message they send or to decompose the messages they receive. Now we give the initial for each participants:

- `A` A knows initially her name, the name of `B`, and `shk(A,B)`, a secret symmetric key pre-shared between the two agents.
- `B` B knows initially his name, the name of `A`, and `shk(A,B)`, a secret symmetric key pre-shared between the two agents.

Actions: This section specifies what messages are exchanged among protocol participants as well as what data is created freshly during a protocol run.

1. `A` generates a nonce `NA` and optionally two text fields `Text1` and `Text2`[2]. Then `A` computes and sends `TokenAB979841` to `B`. On receipt of the message containing `TokenAB979841`, `B` verifies `TokenAB979841` by checking the sequence number, comparing it with the cryptographic check value of the token, thereby verifying the correctness of the distinguishing identifier `B`, if present.

[2]The standard also considers the possibility of using time stamps

Goals: here we specify the goals of the protocol. ISO 9797-4.1 aims at achieving unilateral authentication between the two participants, the claimant `A` and the verifier `B`.

Here we present the second protocol of the ISO 9797-4 specification in SPS based on [Int99].

```
1  Protocol: ISO989742
2
3  Types:
4      Agent A,B;
5      Function hash;
6  Formats:
7      tokenAB979842 (Number,Msg);
8      # Agent is optional in the standard
9      fkab(Number,Agent,Number);
10     farg(SymmetricKey,Msg);
11     fab(Number,Number);
12     fm2(Msg,Number);
13
14 Knowledge:
15     A: A,B,shk(A,B);
16     B: B,A,shk(A,B);
17
18 Actions:
19     B: Number NB,Text1
20     B->A: fab(NB,Text1)
21     A: Number Text3,Text2
22     A->B: fm2(tokenAB979842(Text3,hash(farg(shk(A,B),fkab(NB,B,Text2)))),
           Text2)
23
24 Goals:
25     B authenticates A on Text2
26
27 Private:
28     shk(A,B)
```

Listing 5.16: 9798-4.2 in SPS

The main difference of the two-step protocol, with respect to the one-step version, is the use of a random number instead of a sequence number (or a time stamp). Hence, there is the need of an extra step in the protocol run, to implement the challenge-response mechanism.

5.5.2 ISO/IEC 9798-4 Formats

As defined in ISO/IEC 9798-1, when the result of concatenating two or more data items is an input to a cryptographic check function as part of one of the mechanisms specified in this part of ISO/IEC 9798, this result shall be

composed so that it can be uniquely resolved into its constituent data strings, i.e. so that there is no possibility of ambiguity in interpretation. Since the actual implementation of the formats is application specific, here we do not give a precise description of the formats as in the previous examples but we simply assume that the following formats, and the associated constructor and destructor, satisfy the aforementioned property. The formats for ISO9798-4.1 are:

- `tokenAB979841`: this format models the authentication token. It is a concatenation of three fields; namely a nonce, a text, and a hashed-value. According to the standard, the parsing of these fields of this format(and all other formats) must unambiguous. Therefore, a possible way to achieve that is using fixed-length fields. Following is `tokenAB979841` represented in the notation for formats that we used earlier in PACE and TLS:

$$\texttt{tokenAB979841}(\texttt{nonce}, \texttt{text}, \texttt{hashed}) = \texttt{nonce} \cdot \texttt{text} \cdot \texttt{hashed}$$

- `fkab`: this format models a concatenation of three fields. Based on the standards requirement on formats (must be unambiguous), we can present this format as follows:

$$\texttt{fkab}(\texttt{nonce}, \texttt{agent}, \texttt{text}) = \texttt{nonce} \cdot \texttt{agent} \cdot \texttt{text}$$

- `farg`: this format models the argument of the hash function. It structures two fields: a symmetric key, and a message formatted with `fkab`. Following is its structure:

$$\texttt{farg}(\texttt{key}, \texttt{msg}) = \texttt{key} \cdot \texttt{msg}$$

- `fm1`: this format models the data packet sent over the network, it has three fields: a message formatted with `tokenAB979841`, an agent name, and a text, as follows:

$$\texttt{fm1}(\texttt{msg}, \texttt{agent}, \texttt{text}) = \texttt{msg} \cdot \texttt{agent} \cdot \texttt{text}$$

The formats for ISO9798-4.2 may be implemented using the same style of encoding.

5.5.3 Analysis Results for ISO/IEC 9798-4

Here we present the formal verification results of these protocols.

Proverif

Using the SPS compiler, we translated the ISO9798-4.1 and ISO9798-4.2 specification into an Applied π code. The summary of Proverif analysis of these protocols is shown in Table 5.7.

Table 5.7: Proverif Analysis Summary for ISO9798-4

Goal of ISO9798-4.1	Proverif result
Authentication Goals `B authenticates A on Text1`	Proof found
Goal of ISO9798-4.2	**Proverif result**
Authentication Goals `B authenticates A on Text2`	Proof found

OFMC

SPS compiler also generates AVISPA IF specifications that can be checked with the OFMC. The summary of the OFMC results for ISO9798-4 (1,2) are shown in Table 5.8.

Table 5.8: Proverif Analysis Summary for ISO9798-4

Goal of ISO9798-4.1	OFMC result
Authentication Goals `B authenticates A on Text1`	No attack found
Goal of ISO9798-4.2	**OFMC result**
Authentication Goals `B authenticates A on Text2`	No attack found

5.6 Summary

In this chapter we presented the SPS specifications for some selected protocols that are highly relevant to the electronic identity systems (eID) in general and

FutureID project in particular. The selected protocols were EAC, PACE, TLS, and a selection of ISO/IEC 9798-4 authentication protocols. By means of these examples, we have shown the effectiveness of our approach. In fact, we were able to encode in SPS a significant class of real-world protocols relevant to eID systems. We precisely defined the message formats used in these protocols. We also presented the results of the formal verification that we performed by means of two state of the art verification tools, Proverif and OFMC.

In addition to the four case studies, our test suite that we used to test SPS compiler consists of over 30 protocols. It includes many protocols from the Clark/Jacob library [CJ95, CJ97], and many other widely used protocols like Kerberos, EPMO [GTC+04], Diffie-Hellman, variants of Needham-Schroeder-Lowe, Denning-Sacco,and h530. SPS compiler runs the entire test suite in less than half a minute on a 2.67 GHz machine. As a result for the test, the tools were able to find all known attacks in the suite. Now we summarize the results of running the formal verification tools (Proverif and OFMC) on the auto-generated code from our SPS complier in Table 5.9. A protocol with a star indicates that it is one of our case studies.

Table 5.9: Formal Verification Summary for Protocols Specified in SPS

Protocol	**Proverif**	**OFMC**
Basic-Kerberos	No Attack	No Attack
EPMO	Attack found	Attack found
⋆ EAC	No Attack	No Attack
⋆ PACE	Non-termination	No Attack
IKEv2-DS-PayloadSecrecy	No Attack	No Attack
IKEv2-DS-PayloadAuthB	No Attack	No Attack
IKEv2-DS-KeySecrecy	No Attack	No Attack
IKEv2-DS-KeyAuthB	No Attack	No Attack
AndrewSecureRPCSecrecy	No Attack	No Attack
ISOsymKeyOnePassUnilateralAuthProt	No Attack	No Attack
ISOsymKeyThreePassMutual	No Attack	No Attack
ISOsymKeyTwoPassMutualAuthProt-Corr	No Attack	No Attack
ISOsymKeyTwoPassUnilateralAuthProt	No Attack	No Attack
NonReversible	No Attack	No Attack
⋆ISO9798-4.2	No Attack	No Attack
ISOCCFThreePassMutual	No Attack	No Attack
⋆ISO9798-4.2	No Attack	No Attack
h530	Attack found	Attack found
h530-fix	No Attack	No Attack
SSO	Attack found	Attack found
⋆ tls-noClientAuth	No Attack	No Attack
⋆ tls	No Attack	No Attack
AndrewSecureRPC	Attack found	Attack found
Needham-Schroeder	Attack found	Attack found
Needham-Schroeder-Lowe	No Attack	No Attack
Amended-NSCK	No Attack	No Attack
Denning-Sacco-TimeStamp	No Attack	No Attack
Denning-Sacco-Corr	No Attack	No Attack
NSCK	No Attack	No Attack
ISOpubKeyOnePassUnilateralAuthProt	No Attack	No Attack
ISOpubKeyTwoPassMutualAuthProt-CORR	No Attack	No Attack
ISOpubKeyTwoPassUnilateralAuthProt	No Attack	No Attack

Chapter 6

Related Work

The formal definition of languages based on the Alice-and-Bob notation requires one to identify the concrete set of actions that honest agents have to perform, which is relevant both for a formal model for verification and for generating implementations. Previous works have proposed fairly involved deduction systems for this purpose and there is no (even informal) justification why these systems would be suitable definitions. Our high-level semantics $[\![\cdot]\!]_H$, inspired by [Möd09, CR10], gives a mathematically succinct and uniform definition of Alice-and-Bob notation following a few general principles, and at the same time it supports an arbitrary set of operators and algebraic properties. We think that the succinctness and generality are a strong argument for this semantics as a standard. As $[\![\cdot]\!]_H$ entails problems that are not recursively computable in general, we defined the low-level semantics $[\![\cdot]\!]_L$ for a particular theory and proved its correctness with respect to $[\![\cdot]\!]_H$. While $[\![\cdot]\!]_L$ is similar (and similarly involved) to previous definitions of a semantics for the Alice-and-Bob notation [Low97a, Mil97, CVB06, BN07, JRV00, BKRS15], we are the first to give a complete formal treatment of the key algebraic properties for destructors, verifies, exponentiation and multiplication.

A very recent treatment of Alice and Bob notation [BKRS15] is very similar to our low-level semantics $[\![\cdot]\!]_L$. They translate protocol specifications in Alice and Bob notation to an intermediate format for each role. This intermediate format is detailed with message derivation and checking, making it very similar

to our operational strands. They further translate this intermediate formats to Tamarin [MSCB13], while we translate to implementations in JavaScript and formal models in the input language of ProVerif and OFMC. We believe that both of their works and ours are linkable to other similar tools. However as their work is limited to subterm convergent theories, it cannot handle exponentiation and multiplication as we do here.

With respect to other implementation generators like [Car94, MM01, TH05, Mod12b, Qua13, Mod14], our key improvements are as follows. First, we give a uniform way to generate both formal models and implementation from the operational strands, ensuring a one-to-one correspondence between them. Second, replacing the abstract concatenation operator from formal models with formats allows us to generate code for any real-world structuring mechanism like XML formats or TLS-style messages. The only work that provides similar features is [BBH12], which however starts at the π calculus level, comparable to the output of our low-level semantics. In reference to works that consider the verification of the actual implementation source code like [BFCZ12], we agree with [BFGT06] that the converse problem, i.e., turning formal models into code like in this thesis, is harder. However, in the case of SPS this extra effort removes a large part of the burden from the user, i.e., SPS carries the task of formally verifiable implementations to a higher level of abstraction without suffering from flaws that are abstracted away in the formal model.

Finally, we point out a strong similarity between our notion of knowledge and the notion of *frames* in Applied π calculus [AF01]. We allow ourselves minor deviations from the frame concept, in particular not using *name restrictions*; instead, constants are by default not public in our setting. This makes our treatment easier but does not fundamentally change the concept (or its expressive power). For what concerns existing decidability results for frames, the deduction relation $\vdash$ has been studied, e.g., in [AC06]. It is known that deduction is decidable for convergent subterm theories (like our equations (1)–(8)) and that disjoint associative-commutative operators as in (9)-(11) can easily be combined with it. Note that with equation (9) that links exponentiation and multiplication we have another property that is crucial for any use of Diffie-Hellman protocols and that is not covered by existing results. Further, many results consider the static equivalence of frames which is interesting for privacy properties, namely whether the intruder is able to distinguish two frames ("knowledges"). In the SPS semantics, we have a substantially different problem to solve: we have only one knowledge M (and it is the knowledge of an honest agent) and we need to finitely characterize $ccs(M)$, i.e., what checks the agent can make on M to ensure that all received messages have the required shape. This indeed has some similar traits to static equivalence: also here one has to check pairs of recipes (albeit with respect to two frames). Despite this similarity, the problems are so different that it seems not directly possible to re-use decision procedures

for static equivalence for computing $ccs(M)$. Moreover, our `exp`/`mult` theory is not yet supported in static equivalence results. A further investigation and generalization, namely with inverses for `mult`, is part of our ongoing research.

Part II

Protocol Typing and Composition

Chapter 7

Introduction

Relative soundness results have proved helpful in the automated verification of security protocols. Such results allow for the reduction of a complex verification problem into a simpler one, if the protocol in question satisfies sufficient conditions. We are in particular interested in conditions that are of a syntactic nature, i.e., can be established without an exploration of the state space of the protocol.

A first kind of such results are *typing results* [HLS03, BP05, Möd12a, AD14]. Here we consider a *typed model*, a restriction of the standard protocol model in which honest agents do not accept any ill-typed messages. This may seem unreasonable at first sight, since in the real-world agents have no way to tell the type of a random bitstring, let alone distinguish it from the result of a cryptographic operation; yet in our model, they accept only well-typed messages. The relative soundness of such a typed model means that if the protocol has an attack, then it also has a well-typed attack. This does not mean that the intruder cannot send ill-typed messages, but rather that this does not give him any advantage as he could perform a "similar" attack with only well-typed messages. Thus, if we are able to verify that a protocol is secure in the typed model, then it is secure also in an untyped model. Typically, the conditions sufficient to achieve such a result are that all composed message patterns of the protocol have a different (intended) type that can somehow be distinguished, e.g., by a tag. The restriction to a typed model in some cases yields a decidable verification

problem [Low99, MSDL99, CKR+03, RS03], and allows for the application of more tools and often significantly reduces verification time in practice [AC04, BP05].

A similar kind of relative soundness results appears in *compositional reasoning*. We consider in this thesis the *parallel composition* of protocols, i.e., running two protocols over the same communication medium, and these protocols may use, e.g., the same long-term public keys. (In the case of disjoint cryptographic material, compositional reasoning is relatively straightforward.) The compositionality result is to show that if two protocols satisfy their security goals in isolation, then their parallel composition is secure, given that the protocols meet certain sufficient conditions. Thus, it suffices to verify the protocols in isolation. The sufficient conditions in this case are similar to the one in the typing result: every composed message can be uniquely attributed to one of the two protocols, which again may be achieved, e.g., by tags.

Organization

In this part of the thesis, we move to a slightly different model, i.e., instead of working with a set of operators with algebraic properties (as we do in the first part), from now on we work in the free algebra: the initial term algebra without any algebraic properties. However, we are still able to handle many desired features in security protocols like combined keys, freshly created public/private key pairs, different message formats and thus generalize over similar work like [AD07]. This enables us to reason not only about Clark and Jacob library [CJ97], but also many other well-known protocols like TLS, and Kerberos. In the rest of this chapter, we introduce the message and intruder models in the free algebra. In Chapter 8, we introduce a symbolic protocol model based on strands and properties in the geometric fragment. We also show how to reduce the verification of the security properties to solving constraints, for which we give a sound, complete and terminating reduction calculus. In Chapter 9, we give our typing and parallel compositionality results, and we introduce a tool that automatically checks if protocols are parallel-composable and report about our experimental results.

7.1 Message Model

Let $\hat{\Sigma}$ be a finite set of *operators* (also referred to as *function symbols*) that are available to all agents, including the intruder. Table 7.1 shows a concrete example of $\hat{\Sigma}$ that is representative for a wide range of security protocols. Further, let $\mathcal{C}$ be a countable set of *constants* and $\mathcal{V}$ a countable set of *variables*, such that $\hat{\Sigma}$, $\mathcal{V}$ and $\mathcal{C}$ are pairwise disjoint. We write $\mathcal{T}_{\hat{\Sigma}\cup\mathcal{C}}(\mathcal{V})$ for the set of *terms* built with these constants, variables and operators, and $\mathcal{T}_{\hat{\Sigma}\cup\mathcal{C}}$ for the set of *ground terms*. We call a term t *atomic* (and write $atomic(t)$) if $t \in \mathcal{V} \cup \mathcal{C}$ or for some term $s, t = \mathsf{pub}(s)$, and *composed* otherwise. We use also other standard notions such as *subterm*, denoted by $\sqsubseteq$, and *substitution*, denoted by σ, as we did in the first part of this thesis.

The set of constants $\mathcal{C}$ is partitioned into three countable and pairwise disjoint subsets: (i) the set $\mathcal{C}_{P_i}$ of *short-term constants* for each protocol P_i, denoting the constants that honest agents freshly generate in P_i; (ii) the set $\mathcal{C}_{priv}$ of *long-term secret constants*; and (iii) the set $\mathcal{C}_{pub}$ of *long-term public constants*. This partitioning will be useful for compositional reasoning: roughly speaking, we will allow the intruder to obtain all public constants, and define that it is an attack if the intruder finds out any of the secret constants.

Formats: Revisited

We continue using the notion of formats that we introduced earlier. This notion is crucial to make our typing and compositionality results applicable to real-world protocols like TLS. By using formats as we explained earlier, we break with the formal-methods tradition of representing clear-text structures of data by a *pair* operator $(\cdot, \cdot)$. For instance, a typical specification may contain expressions like (A, NA) and $(NB, (KB, A))$. This representation neglects the details of a protocol implementation that may employ various mechanisms to enable a receiver to decompose a message in a unique way (e.g., field-lengths or XML-tags). The abstraction has the disadvantage that it may easily lead to false positives and false negatives. For example, the two messages above have a unifier $A \mapsto NB$ and $NA \mapsto (KB, NA)$, meaning that a message meant as (A, NA) may accidentally be parsed as $(NB, (KB, A))$, which could lead to a "type-flaw" attack. This attack may, however, be completely unrealistic.

To handle this, previous typing results have used particular *tagging schemes*, e.g., requiring that each message field starts with a tag identifying the type of that field. Similarly, compositionality results have often required that each encrypted message of a protocol starts with a tag identifying the protocol that

Description	Operator	Analysis rule
Symmetric encryption	$\mathtt{scrypt}(\cdot,\cdot)$	$\mathtt{Ana}(\mathtt{scrypt}(k,m)) = (\{k\},\{m\})$
Asymmetric encryption	$\mathtt{crypt}(\cdot,\cdot)$	$\mathtt{Ana}(\mathtt{crypt}(\mathsf{pub}(t),m)) = (\{t\},\{m\})$
Signature	$\mathtt{sign}(\cdot,\cdot)$	$\mathtt{Ana}(\mathtt{sign}(t,m)) = (\emptyset,\{m\})$
Formats, e.g., $\mathtt{f}_1$	$\mathtt{f}_1(t_1,\cdots,t_n)$	$\mathtt{Ana}(\mathtt{f}_1(t_1,\cdots,t_n)) = (\emptyset,\{t_1,\cdots,t_n\})$
One-way functions, e.g., hash	$\mathsf{hash}(\cdot)$	$\mathtt{Ana}(\mathsf{hash}(t)) = (\emptyset,\emptyset)$
Public key of a given private key	$\mathsf{pub}(\cdot)$	$\mathtt{Ana}(\mathsf{pub}(t)) = (\emptyset,\emptyset)$
All other terms		$\mathtt{Ana}(t) = (\emptyset,\emptyset)$

Table 7.1: Example Operators $\hat{\Sigma}$

this message was meant for. Besides the fact that this does not really solve the problem of false positives and false negatives due to the abstraction, practically no existing protocol uses exactly this schema. Moreover, it is completely unrealistic to think that a widely used protocol like TLS would be changed just to make it compatible with the assumptions of an academic work — the only chance to have it changed is to point out a vulnerability that can be fixed by the change.

Formats are a means to have a faithful yet abstract model. We define formats as functions from data-packets to concrete strings. In Chapter 5, we already explained various formats for TLS and other protocols. For example, a format from TLS is **client_hello(time, random, session_id, cipher_suites, comp_methods)** $= \mathsf{byte}(1)\cdot\mathsf{off}_3(\mathsf{byte}(3)\ \cdot\mathsf{byte}(3)\cdot \mathit{time}\cdot\mathit{random}\cdot\mathsf{off}_1(\mathit{session\_id})\cdot\ \mathsf{off}_2(\mathit{cipher\_suites})\cdot \mathsf{off}_1(\mathit{comp\_methods}))$, where $\mathsf{byte}(n)$ means one concrete byte of value n, $\mathsf{off}_k(m)$ means that m is a message of variable length followed by a field of k bytes, and $\cdot$ represents string concatenation.

In the abstract model, we are going to use only abstract terms like the part in bold in the above example. It is shown in [MK14] that under certain conditions on formats this abstraction introduces neither false positives nor false negatives. The conditions are essentially that formats must be parsed in an unambiguous way and must be pairwise disjoint; then every attack on the concrete bytestring model can be simulated in the model based on abstract format symbols (in the free algebra). Both in typing and compositionality, these conditions allow us to apply our results to real world protocols no matter what formatting scheme they actually use (e.g., a TLS message cannot be accidentally be parsed as an EAC message). In fact, these reasonable conditions are satisfied by many protocols in practice, and whenever they are violated, typically we have a good chance to find actual vulnerabilities. As we explained in Part I, formats are *transparent* in the sense that if the intruder learns $\mathtt{f}(t_1,\ldots,t_n)$, then he also obtains the t_i.

7.2 Intruder Model

We specify how the intruder can compose and decompose messages in the style of the Dolev-Yao intruder model as we have in Definition 3.3.

Definition 7.1 An *intruder knowledge* $\mathcal{M}$ is a finite set of ground terms $t \in \mathcal{T}_{\hat{\Sigma}\cup\mathcal{C}}$. Let $\texttt{Ana}(t) = (K, T)$ be a function that returns for every term t a pair (K, T) of finite sets of subterms of t. We define $\Vdash$ to be the least relation between a knowledge $\mathcal{M}$ and a term t that satisfies the following *intruder deduction rules*:

$$\frac{}{\mathcal{M} \Vdash t} \; t \in \mathcal{M} \quad \text{(Axiom)},$$

$$\frac{}{\mathcal{M} \Vdash c} \; c \in \mathcal{C}_{pub} \quad \text{(Public)},$$

$$\frac{\mathcal{M} \Vdash t_1 \quad \cdots \quad \mathcal{M} \Vdash t_n}{\mathcal{M} \Vdash f(t_1, \cdots, t_n)} \; f \in \hat{\Sigma}^n \quad \text{(Compose)},$$

$$\frac{\mathcal{M} \Vdash t \quad \mathcal{M} \Vdash k_1 \quad \cdots \quad \mathcal{M} \Vdash k_n}{\mathcal{M} \Vdash t_i} \; \texttt{Ana}(t) = (K, T),\; K = \{k_1, \cdots, k_n\},\; t_i \in T \quad \text{(Decompose)},$$

The rules (Axiom) and (Public) formalize that the intruder can derive any term $t \in \mathcal{M}$ that is in his knowledge and every long-term public constant $c \in \mathcal{C}_{pub}$, respectively. The (Compose) rule formalizes that he can compose known terms with any operator in $\hat{\Sigma}$ (where n denotes the arity of f). Table 7.1 provides an example $\hat{\Sigma}$ for standard cryptographic operators.

Table 7.1 also defines the function `Ana` for each of $\hat{\Sigma}$ operators. We rely on this function to define message decomposition; as it encodes the analysis rules for terms in the form of $\texttt{Ana}(t) = (K, T)$, which intuitively says that if the intruder knows the keys in set K, then he can analyze the term t and obtain the set of messages T. We require that all elements of K and T are subterms of t (without any restriction, the relation $\Vdash$ would be undecidable). Based on `Ana`, the generic (Decompose) deduction rule formalizes how the intruder decomposes his messages: that for any term with an `Ana` rule, if the intruder can derive the keys in K, he can also derive all the subterms of t in T.

Consider, e.g., the analysis rule for symmetric encryption given in Table 7.1: $\texttt{Ana}(\texttt{scrypt}(k, m)) = (\{k\}, \{m\})$ says that given a term $\texttt{scrypt}(k, m)$ one needs the key k to derive m. By default, atomic terms cannot be analyzed, i.e., $\texttt{Ana}(t) = (\emptyset, \emptyset)$.

7.3 Summary

In this chapter, we considered a slightly different model that works in the free algebra. Therefore, we defined a message model in which we specified the set of operators $\hat{\Sigma}$ to build messages. We also explained the rationale behind the partitioning of constants and how formats relate to our model. Then, we introduced the intruder model that explains how he derives his messages. In the next chapter, we define a symbolic model for protocols based on the previous definitions.

Chapter 8

Symbolic Protocol Model

We define a protocol by a set of operational strands and a set of *goal predicates* that the protocol is supposed to achieve. The semantics of a protocol is an infinite-state transition system over symbolic states and security means that all reachable states satisfy the goal predicates. A *symbolic state* $(\mathcal{S}; \mathcal{M}; E; \phi)$ consists of a set $\mathcal{S}$ of operational strands (representing the honest agents), the intruder knowledge $\mathcal{M}$, a set E of events that have occurred, and a symbolic constraint ϕ on the free variables occurring in the state. We first define the symbolic constraints in Section 8.1. Then, we revisit the operational strands in Section 8.2 in order to define the transition relation on symbolic states. In Section 8.3 we define protocol goals in the geometric fragment and how we translate them to symbolic constraints. Finally, we explain how to solve these constraints in Section 8.4.

8.1 Symbolic Constraints

The syntax of *symbolic constraints* is

$$\phi \;:=\; \mathcal{M} \Vdash t \mid \phi_\sigma \mid \neg\exists \bar{x}.\, \phi_\sigma \mid \phi \wedge \phi \mid \underbrace{\phi \vee \phi \mid \exists \bar{x}.\, \phi}_{\star} \quad \text{with} \quad \phi_\sigma \;:=\; s \doteq t \mid \phi_\sigma \wedge \phi_\sigma$$

where s, t range over terms in $\mathcal{T}_{\hat{\Sigma}\cup\mathcal{C}}(\mathcal{V})$, $\mathcal{M}$ is a finite set of terms (not necessarily ground) and $\bar{x}$ is a list of variables. The sub-language ϕ_σ defines *equations* on messages, and we can existentially quantify variables in them, e.g., consider a ϕ of the form $\exists x.\, y \doteq f(x)$. We refer to equations also as *substitutions* since the application of the standard most general unifier on a conjunction of equations results in a set of substitutions. The constraints can contain such substitutions in positive and negative form (excluding all instances of a particular substitution).

$\mathcal{M} \Vdash t$ is an *intruder constraint*: the intruder must be able to derive term t from knowledge $\mathcal{M}$. Note that we have no negation at this level, i.e., we cannot write negated intruder constraints. A *base constraint* is a constraint built according to this grammar without the two cases marked $\star$, i.e., disjunction $\phi \vee \phi$ and existential quantification $\exists \bar{x}.\, \phi$. The latter may only occur in negative substitutions.

For ease of writing, we define the semantics of the constraint language as standard for each construct (rather than following strictly the grammar of ϕ).

Definition 8.1 Given an interpretation $\mathcal{I}$, which maps each variable in $\mathcal{V}$ to a ground term in $\mathcal{T}_{\hat{\Sigma}}$, and a symbolic constraint ϕ, the model relation $\mathcal{I} \models \phi$ is:

$\mathcal{I} \models \mathcal{M} \Vdash t$	iff	$\mathcal{I}(\mathcal{M}) \Vdash \mathcal{I}(t)$
$\mathcal{I} \models s \doteq t$	iff	$\mathcal{I}(s) = \mathcal{I}(t)$
$\mathcal{I} \models \neg\phi$	iff	not $\mathcal{I} \models \phi$
$\mathcal{I} \models \phi_1 \wedge \phi_2$	iff	$\mathcal{I} \models \phi_1$ and $\mathcal{I} \models \phi_2$
$\mathcal{I} \models \phi_1 \vee \phi_2$	iff	$\mathcal{I} \models \phi_1$ or $\mathcal{I} \models \phi_2$
$\mathcal{I} \models \exists x.\phi$	iff	there is a term $t \in \mathcal{T}_{\hat{\Sigma}}$ such that $\mathcal{I}[x \mapsto t] \models \phi$

We say that $\mathcal{I}$ is a *model* of ϕ iff $\mathcal{I} \models \phi$, and that ϕ is *satisfiable* iff it has a model. Two constraints are *equivalent*, denoted by $\equiv$, iff they have the same models. We define as standard the *variables* (denoted by $var(\cdot)$) and the *free variables* (denoted by $fv(\cdot)$) of terms, sets of terms, equations, and constraints. A constraint ϕ is *closed*, in symbols $closed(\phi)$, iff $fv(\phi) = \emptyset$.

Every constraint ϕ can be quite straightforwardly transformed into an equivalent constraint of the form

$$\phi \;\equiv\; \exists \bar{x}.\, \phi_1 \vee \ldots \vee \phi_n\,,$$

where the ϕ_i are base constraints. Unless noted otherwise, in the following we will assume that constraints are in this form.

Definition 8.2 A constraint is *well-formed* if each of its base constraints ϕ_i satisfies the following condition: we can order the conjuncts of ϕ_i such that $\phi_i = \mathcal{M}_1 \Vdash t_1 \wedge \ldots \wedge \mathcal{M}_n \Vdash t_n \wedge \phi_i'$, where ϕ_i' contains no further $\Vdash$ constraints and such that $\mathcal{M}_j \subseteq \mathcal{M}_{j+1}$ (for $1 \leq j < n$) and $fv(\mathcal{M}_j) \subseteq fv(t_1) \cup \ldots \cup fv(t_{j-1})$.

Intuitively, this condition expresses that the intruder knowledge grows monotonically and all variables that appear in an intruder knowledge constraint occur in a term that the intruder sent earlier in the protocol execution. We will ensure that all constraints that we deal with are well-formed.

8.2 Operational Strands: Revisited

In Chapter 2, we introduced the operational strands as the target language of a translation that defined the semantics of SPS. Here, we use it again to define the protocol semantics, but we exclude the initial knowledge from operational strands and include positive and negative quantified equations on messages. More precisely, we consider the syntax of operational strands as follows:

$$S \;:=\; \mathsf{send}(t).S \mid \mathsf{receive}(t).S \mid \mathsf{event}(t).S \mid (\exists \bar{x}.\, \phi_\sigma).S \mid (\neg\exists \bar{x}.\phi_\sigma).S \mid 0$$

where ϕ_σ is as defined above; we will omit the parentheses when there is no risk of confusing the dots. *fv* and *closed* extend to operational strands as expected, with the exception of the receiving step, which can bind variables: we set $fv(\mathsf{receive}(t).S) = fv(S) \setminus fv(t)$. According to the semantics that we define below, in $\mathsf{receive}(x).\mathsf{receive}(f(x)).\mathsf{send}(x).0$ the variable x is bound actually in the first receive, i.e., the strand is equivalent to $\mathsf{receive}(x).\mathsf{receive}(y).(y \doteq f(x)).\mathsf{send}(x).0$.

A *symbolic state* $(\mathcal{S}; \mathcal{M}; E; \phi)$ consists of a (finite or countable) set[1] $\mathcal{S}$ of closed operational strands, a finite set $\mathcal{M}$ of terms representing the intruder knowledge, a finite set E of events, and a formula ϕ. *fv* and *closed* extend to symbolic states again as expected. We ensure that $fv(\mathcal{S}) \cup fv(\mathcal{M}) \cup fv(E) \subseteq fv(\phi)$ for all reachable states $(\mathcal{S}; \mathcal{M}; E; \phi)$ and that ϕ is well-formed. This is so since in the transition system defined below, the operational strands of the initial state are closed and the transition relation only adds new variables in the case of $\mathsf{receive}(t)$, but in this case ϕ is updated with $\mathcal{M} \Vdash t$.

A *protocol specification* $(\mathcal{S}_0, G)$ (or simply *protocol*) consists of a set $\mathcal{S}_0$ of closed operational strands and a set G of goal predicates (defined below). For simplicity, we assume that the bound variables of any two different strands in $\mathcal{S}_0$ are disjoint (which can be achieved by α-renaming). The strands in $\mathcal{S}_0$ induce an *infinite-state transition system* with *initial state* $(\mathcal{S}_0; \emptyset; \emptyset; \top)$ and a transition relation $\Rightarrow$ defined as the least relation closed under six transition rules:

T1 $(\{\mathsf{send}(t).S\} \cup \mathcal{S}; \mathcal{M}; E; \phi) \Rightarrow (\{S\} \cup \mathcal{S}; \mathcal{M} \cup \{t\}; E; \phi)$

[1] Some approaches instead use multi-sets as we may have several identical strands, but since one can always make a strand unique, using sets is without loss of generality.

T2 $(\{\mathsf{receive}(t).S\} \cup \mathcal{S}; \mathcal{M}; E; \phi) \Rightarrow (\{S\} \cup \mathcal{S}; \mathcal{M}; E; \phi \wedge \mathcal{M} \Vdash t)$

T3 $(\{\mathsf{event}(t).S\} \cup \mathcal{S}; \mathcal{M}; E; \phi) \Rightarrow (\{S\} \cup \mathcal{S}; \mathcal{M}; E \cup \mathsf{event}(t); \phi)$

T4 $(\{\phi'.S\} \cup \mathcal{S}; \mathcal{M}; E; \phi) \Rightarrow (\{S\} \cup \mathcal{S}; \mathcal{M}; E; \phi \wedge \phi')$

T5 $(\{0\} \cup \mathcal{S}; \mathcal{M}; E; \phi) \Rightarrow (\mathcal{S}; \mathcal{M}; E; \phi)$

T6 $(\mathcal{S}; \mathcal{M}; E; \phi) \Rightarrow (\mathcal{S}; \mathcal{M}; E \cup \{lts(c)\}; \phi)$ for every $c \in \mathcal{C}_{priv}$

The rule T1 formalizes that sent messages are added to the intruder knowledge $\mathcal{M}$. T2 formalizes that an honest agent receives a message of the form t, and that the intruder must be able to create that message from his current knowledge, expressed by the new constraint $\mathcal{M} \Vdash t$; this indirectly binds the free variables of the rest of the strand in the sense that they are now governed by the constraints of the state. In a non-symbolic model, one would at this point instead need to consider all ground instances of t that the intruder can generate. T3 formalizes that we add events to the set E. T4 simply adds the constraint ϕ' to the constraint ϕ. T5 says that if a strand reaches it end 0, then we remove it. Finally, for every secret constant c in $\mathcal{C}_{priv}$, T6 adds the event $lts(c)$ to the set E; to indicate that c is a long term secret. (We define later as a goal that the intruder never obtains any c for which $lts(c) \in E$.) We cannot have this in the initial set E as we need it to be finite; this construction is later crucial in the parallel composition proof as we can at any time blame a protocol (in isolation) that leaks a secret constant. We discuss below that in practice this semantical rule does not cause trouble to the verification of the individual protocols.

8.3 Goal Predicates in the Geometric Fragment

We express goals by *state formulas in the geometric fragment* [Gut14]. These formulas can refer to the intruder knowledge, but in a restricted manner so that we obtain constraints of the form ϕ. Security then means: every reachable state in the transition system induced by $\mathcal{S}_0$ satisfies each state formula, and thus an *attack* is a reachable state where at least one goal does not hold.

The constraints ϕ we have defined above are interpreted only with respect to an interpretation of the free variables, whereas the state formulas are evaluated with respect to a symbolic state, including the current intruder knowledge and events that have occurred (as before, we define the semantics for each construct).

Definition 8.3 *State formulas* Ψ *in the geometric fragment* are defined as:

$$\Psi := \forall \bar{x}.\,(\psi \implies \psi_0) \text{ with } \begin{cases} \psi := \mathsf{ik}(t) \mid \mathsf{event}(t) \mid t \doteq t' \mid \psi \wedge \psi' \mid \psi \vee \psi' \mid \exists \bar{x}.\psi \\ \psi_0 := \mathsf{event}(t) \mid t \doteq t' \mid \psi_0 \wedge \psi_0' \mid \psi_0 \vee \psi_0' \mid \exists \bar{x}.\psi_0 \end{cases}$$

where $\mathsf{ik}(t)$ denotes that the intruder knows the term t. $fv(\cdot)$ and *closed* extend to state formulas as expected. Given a state formula Ψ, an interpretation $\mathcal{I}$, and a state $\mathfrak{S} = (\mathcal{S}; \mathcal{M}; E; \phi)$, we define $\mathcal{I}, \mathcal{M}, E \models_{\mathfrak{S}} \Psi$ as:

$$\begin{array}{lll}
\mathcal{I}, \mathcal{M}, E \models_{\mathfrak{S}} \mathsf{event}(t) & \text{iff} & \mathcal{I}(\mathsf{event}(t)) \in \mathcal{I}(E) \\
\mathcal{I}, \mathcal{M}, E \models_{\mathfrak{S}} \mathsf{ik}(t) & \text{iff} & \mathcal{I}(\mathcal{M}) \Vdash \mathcal{I}(t) \\
\mathcal{I}, \mathcal{M}, E \models_{\mathfrak{S}} s \doteq t & \text{iff} & \mathcal{I}(s) = \mathcal{I}(t) \\
\mathcal{I}, \mathcal{M}, E \models_{\mathfrak{S}} \Psi \wedge \Psi' & \text{iff} & \mathcal{I}, \mathcal{M}, E \models_{\mathfrak{S}} \Psi \text{ and } \mathcal{I}, \mathcal{M}, E \models_{\mathfrak{S}} \Psi' \\
\mathcal{I}, \mathcal{M}, E \models_{\mathfrak{S}} \Psi \vee \Psi' & \text{iff} & \mathcal{I}, \mathcal{M}, E \models_{\mathfrak{S}} \Psi \text{ or } \mathcal{I}, \mathcal{M}, E \models_{\mathfrak{S}} \Psi' \\
\mathcal{I}, \mathcal{M}, E \models_{\mathfrak{S}} \neg\Psi & \text{iff} & \text{not } \mathcal{I}, \mathcal{M}, E \models_{\mathfrak{S}} \Psi \\
\mathcal{I}, \mathcal{M}, E \models_{\mathfrak{S}} \exists x.\,\Psi & \text{iff} & \text{there exists } t \in \mathcal{T}_{\hat{\Sigma}} \text{ and } \mathcal{I}[x \mapsto t] \models_{\mathfrak{S}} \Psi
\end{array}$$

Definition 8.4 A protocol $P = (\mathcal{S}_0, \{\Psi_0, \cdots, \Psi_n\})$, where the Ψ_i are closed state formulas, *has an attack against goal* Ψ_i iff there exist a reachable state $\mathfrak{S} = (\mathcal{S}; \mathcal{M}; E; \phi)$ in the transition system induced by $\mathcal{S}_0$ and an interpretation $\mathcal{I}$ such that $\mathcal{I}, \mathcal{M}, E \models_{\mathfrak{S}} \neg\Psi_i$ and $\mathcal{I} \models \phi$. We also call $\mathfrak{S}$ an *attack state* in this case.

Note that in this definition the interpretation $\mathcal{I}$ does not matter in $\mathcal{I}, \mathcal{M}, E \models_{\mathfrak{S}} \neg\Psi_i$ because Ψ_i is closed.

Example 8.1 *If a protocol generates the event* $\mathsf{secret}(x_A, x_B, x_m)$[2] *to denote that the message* x_m *is supposed to be a secret between agents* x_A *and* x_B*, and—optionally—the event* $\mathsf{release}(x_m)$ *to denote that* x_m *is no longer a secret, then we can formalize* secrecy *via the state formula*

$$\forall x_A x_B x_m.(\mathsf{secret}(x_A, x_B, x_m) \wedge \mathsf{ik}(x_m) \implies x_A = \mathsf{i} \vee x_B = \mathsf{i} \vee \mathsf{release}(x_m))$$

where i *denotes the intruder. The release event can be used to model declassification of secrets as needed to verify perfect forward secrecy (when other data should remain secret even under the release of temporary secrets). We note that previous compositionality approaches do not support such goals. A typical formulation of* non-injective agreement *[Low97b] uses the two events* $\mathsf{commit}(x_A, x_B, x_m)$*, which represents that* x_A *intends to send message* x_m *to*

[2] Recall that we should write this as $\mathsf{event}(\mathsf{secret}(x_A, x_B, x_m))$, but, for readability, here and below we will omit the outer $\mathsf{event}(\cdot)$ when it is clear from context.

x_B), and $\mathsf{running}(x_A, x_B, x_m, x_C)$*, which represents that x_B believes to have received x_m from x_A, with x_C a unique identifier:*

$$\forall x_A x_B x_m x_C.\, (\mathsf{running}(x_A, x_B, x_m, x_C) \implies \mathsf{commit}(x_A, x_B, x_m) \vee x_A = \mathsf{i} \vee x_B = \mathsf{i})$$

and injective agreement *would additionally require:*

$$\forall x_A x_B x_m x_C x'_C.\, \mathsf{running}(x_A, x_B, x_m, x_C) \wedge \mathsf{running}(x_A, x_B, x_m, x'_C) \implies x_A = \mathsf{i} \vee x_B = \mathsf{i} \vee x_C = x'_C$$

□

8.4 Constraint Solving

We first show how to translate every state formula Ψ in the geometric fragment for a given symbolic state $\mathfrak{S} = (\mathcal{S}; \mathcal{M}; E; \phi)$ into a constraint ϕ' (in the fragment defined in Section 8.1) so that the models of $\phi \wedge \phi'$ represent exactly all concrete instances of $\mathfrak{S}$ that violate Ψ. Then, we extend a rule-based procedure to solve ϕ-style constraints (getting them into an equivalent simple form). This procedure provides the basis for our typing and parallel composition results.

From geometric fragment to symbolic constraints

Consider a reachable symbolic state $(\mathcal{S}; \mathcal{M}; E; \phi)$ and a goal formula Ψ. As mentioned earlier, we require that Ψ is closed. Let us further assume that the bound variables of Ψ are disjoint from the variables (bound or free) of $\mathcal{S}$, $\mathcal{M}$, E, and ϕ. We now define a *translation function* $tr_{\mathcal{M},E}(\Psi) = \phi'$ where ϕ' represents the negation of Ψ with respect to intruder knowledge $\mathcal{M}$ and events E. The negation is actually manifested in the first line of the definition:

$$\begin{array}{lcl}
tr_{\mathcal{M},E}(\forall \bar{x}.\, \psi \Rightarrow \psi_0) & = & \exists \bar{x}.\, tr'_{\mathcal{M},E}(\psi) \wedge tr'_{\mathcal{M},E}(\neg\psi_0) \\
tr'_{\mathcal{M},E}(\mathsf{ik}(t)) & = & \mathcal{M} \Vdash t \\
tr'_{\mathcal{M},E}(\mathsf{event}(t)) & = & \bigvee_{\mathsf{event}(s) \in E} s \doteq t \\
tr'_{\mathcal{M},E}(s \doteq t) & = & s \doteq t \\
tr'_{\mathcal{M},E}(\psi_1 \vee \psi_2) & = & tr'_{\mathcal{M},E}(\psi_1) \vee tr'_{\mathcal{M},E}(\psi_2) \\
tr'_{\mathcal{M},E}(\psi_1 \wedge \psi_2) & = & tr'_{\mathcal{M},E}(\psi_1) \wedge tr'_{\mathcal{M},E}(\psi_2) \\
tr'_{\mathcal{M},E}(\exists \bar{x}.\psi) & = & \exists \bar{x}.tr'_{\mathcal{M},E}(\psi) \\
tr'_{\mathcal{M},E}(\neg\mathsf{event}(t)) & = & \bigwedge_{\mathsf{event}(s) \in E} \neg s \doteq t \\
tr'_{\mathcal{M},E}(\neg s \doteq t) & = & \neg s \doteq t \\
tr'_{\mathcal{M},E}(\neg(\exists \bar{x}.\psi_1 \vee \psi_2)) & = & tr'_{\mathcal{M},E}(\neg\exists \bar{x}.\psi_1) \wedge tr'_{\mathcal{M},E}(\neg\exists \bar{x}.\psi_2) \\
tr'_{\mathcal{M},E}(\neg\neg\phi) & = & tr'_{\mathcal{M},E}(\phi) \\
tr'_{\mathcal{M},E}(\neg\exists \bar{x}.\mathsf{event}(t_1) \wedge \cdots \wedge \mathsf{event}(t_n) \wedge u_1 \doteq v_1 \wedge \cdots u_m \doteq v_m) = & & \\
\quad \bigwedge_{\mathsf{event}(s_1) \in E \ldots \mathsf{event}(s_n) \in E} \neg\exists \bar{x}.\, (s_1 \doteq t_1 \wedge \cdots \wedge t_n \doteq s_n \wedge u_1 \doteq v_1 \wedge \cdots u_m \doteq v_m) & &
\end{array}$$

Theorem 6 *Let $\mathfrak{S} = (\mathcal{S}; \mathcal{M}; E; \phi)$ be a symbolic state and Ψ a formula in the geometric fragment such that $fv(\Psi) = \emptyset$ and $var(\Psi) \cap var(\phi) = \emptyset$. For all $\mathcal{I} \models \phi$, we have $\mathcal{I}, \mathcal{M}, E \models_{\mathfrak{S}} \neg\Psi$ iff $\mathcal{I} \models tr_{\mathcal{M},E}(\Psi)$. Moreover, if ϕ is well-formed, then so is $\phi \wedge tr_{\mathcal{M},E}(\Psi)$.*

Proof. We first prove, by induction, a corresponding property for the function tr' that is called by the tr function. For that assume we have $\mathcal{I}$, $\mathcal{M}$, E, ϕ, and ψ such that $\mathcal{I} \models \phi$, $var(\phi) \cap var(\psi) = \emptyset$, $fv(\phi) \subseteq fv(tr'_{\mathcal{M},E}(\psi)) \subseteq fv(\phi) \cup fv(\psi)$, $fv(\phi) = var(\mathcal{M}) \cup var(E)$. Also we have that $E = \{\mathsf{event}(s_1), \cdots, \mathsf{event}(s_n)\}$ since E is a finite set. We prove $\mathcal{I}, \mathcal{M}, E \models_{\mathfrak{S}} \psi$ iff $\mathcal{I} \models tr'_{\mathcal{M},E}(\psi)$ by induction on the structure of $tr'_{\mathcal{M},E}(\cdot)$:

- $\mathcal{I}, \mathcal{M}, E \models_{\mathfrak{S}} \mathsf{ik}(t)$ iff $\mathcal{I}(\mathcal{M}) \Vdash \mathcal{I}(t)$ iff $\mathcal{I} \models \mathcal{M} \Vdash t = tr'_{\mathcal{M},E}(\mathsf{ik}(t))$.
- $\mathcal{I}, \mathcal{M}, E \models_{\mathfrak{S}} \mathsf{event}(t)$ iff $\mathcal{I}(\mathsf{event}(t)) \in \mathcal{I}(E)$ iff $\mathcal{I}(t) \in \{\mathcal{I}(s_1), \cdots, \mathcal{I}(s_n))\}$ iff $\mathcal{I}(t) = \mathcal{I}(s_1) \vee \cdots \vee \mathcal{I}(t) = \mathcal{I}(s_n)$ iff $\mathcal{I} \models t \doteq s_1 \vee \cdots \vee t \doteq s_n$ iff $\mathcal{I} \models \bigvee_{\mathsf{event}(s) \in E} s \doteq t = tr'_{\mathcal{M},E}(\mathsf{event}(t))$.
- $\mathcal{I}, \mathcal{M}, E \models_{\mathfrak{S}} \psi_1 \vee \psi_2$ iff $\mathcal{I}, \mathcal{M}, E \models_{\mathfrak{S}} \psi_1$ or $\mathcal{I}, \mathcal{M}, E \models_{\mathfrak{S}} \psi_2$ iff $\mathcal{I} \models tr'_{\mathcal{M},E}(\psi_1)$ or $\mathcal{I} \models tr'_{\mathcal{M},E}\psi_2)$ by induction iff $\mathcal{I} \models tr'_{\mathcal{M},E}(\psi_1) \vee tr'_{\mathcal{M},E}(\psi_2) = tr'_{\mathcal{M},E}(\psi_1 \vee \psi_2)$.
- $\mathcal{I}, \mathcal{M}, E \models_{\mathfrak{S}} s \doteq t$ iff $\mathcal{I}(s) = \mathcal{I}(t)$ iff $\mathcal{I} \models s \doteq t = tr'_{\mathcal{M},E}(s \doteq t)$.
- The other cases follow similarly.

Based on this, we prove $\mathcal{I}, \mathcal{M}, E \models \neg\Psi$ iff $\mathcal{I} \models tr_{\mathcal{M},E}(\Psi)$. Let $\Psi = \forall \bar{x}.\psi \implies \psi_0$. Then, $\mathcal{I} \models tr_{\mathcal{M},E}(\Psi) = \exists \bar{x}.\, tr'_{\mathcal{M},E}(\psi) \wedge tr'_{\mathcal{M},E}(\neg\psi_0)$ iff

- exist $\bar{t}$ such that $\mathcal{I}[\bar{x} \mapsto \bar{t}] \models tr'_{\mathcal{M},E}(\psi)$ and $\mathcal{I}[\bar{x} \mapsto \bar{t}] \models tr'_{\mathcal{M},E}(\neg\psi_0)$ iff
- exist $\bar{t}$ such that $\mathcal{I}[\bar{x} \mapsto \bar{t}], \mathcal{M}, E \models_{\mathfrak{S}} \psi$ and $\mathcal{I}[\bar{x} \mapsto \bar{t}], \mathcal{M}, E \models_{\mathfrak{S}} \neg\psi_0$ iff
- exist $\bar{t}$ such that $\mathcal{I}[\bar{x} \mapsto \bar{t}], \mathcal{M}, E \models_{\mathfrak{S}} \psi \wedge \neg\psi_0$ iff
- $\mathcal{I}, \mathcal{M}, E \models_{\mathfrak{S}} \exists \bar{x}.\psi \wedge \neg\psi_0$ iff
- $\mathcal{I}, \mathcal{M}, E \models_{\mathfrak{S}} \neg\forall \bar{x}.\psi \implies \psi_0 = \neg\Psi$.

The well-formedness follows from the fact that in each state, the knowledge $\mathcal{M}$ is a superset of every $\mathcal{M}'$ that occur in a deduction constraint $\mathcal{M}' \Vdash t$ in ϕ. Further, $\mathcal{M}$ can only contain variables that occur in some t for which $\mathcal{M}' \Vdash t$ occurs in ϕ. Thus, $tr_{\mathcal{M},E}(\Psi) \wedge \phi$ is well-formed, if ϕ is. □

Constraint Reduction

As mentioned before, we can transform any well-formed constraint into the form $\phi \equiv \exists \bar{x}.\phi_0 \vee \ldots \vee \phi_n$, where ϕ_i are base constraints, i.e., without disjunction and existential quantification (except in negative substitutions). We now discuss how to find the solutions of such well-formed base constraints. Solving intruder constraints has been studied quite extensively, e.g., in [MS01, RT03, CDM11, Möd12a], where the main application of constraints was for efficient protocol verification for a bounded number of sessions of honest agents. Here, we use constraints rather as a proof argument for the shape of attacks. Our result is of course not restricted to a bounded number of sessions as we do not rely on an exploration of reachable symbolic states (that are indeed infinite) but rather make an argument about the constraints in each of these states.

We consider *constraint reduction rules* of the form

$$\frac{\phi'}{\phi}\ (name),\ cond$$

expressing that ϕ' entails ϕ (if the side condition *cond* holds). However, we will usually read the rule backwards, i.e., as: one way to solve ϕ is to solve ϕ'.

Definition 8.5 The *satisfiability calculus for the symbolic intruder* comprises the following constraint reduction rules:

$$\frac{eq(\sigma) \wedge \sigma(\phi)}{\mathcal{M} \Vdash t \wedge \phi}\ (Unify),\ s,t \notin \mathcal{V}, s \in \mathcal{M},\ \sigma \in mgu(s \doteq t)$$

$$\frac{eq(\sigma) \wedge \sigma(\phi)}{s \doteq t \wedge \phi}\ (Equation),\ \sigma \in mgu(s \doteq t),\ s \notin \mathcal{V}\ or\ s \in fv(t) \cup fv(\phi)$$

$$\frac{\phi}{\mathcal{M} \Vdash c \wedge \phi}\ (PubConsts),\ c \in \mathcal{C}_{pub}$$

$$\frac{\mathcal{M} \Vdash t_1, \cdots, \mathcal{M} \Vdash t_n}{\mathcal{M} \Vdash f(t_1, \cdots, t_n)}\ (Compose),\ f \in \hat{\Sigma}^n$$

$$\frac{\bigwedge_{k \in K} \mathcal{M} \Vdash k \wedge (\mathcal{M} \Vdash t \wedge \phi)^{T \gg \mathcal{M}}}{\mathcal{M} \Vdash t \wedge \phi}\ (Decompose),\ s \in \mathcal{M}, \mathtt{Ana}(s) = (K,T), T \not\subseteq \mathcal{M},\ \text{and } (Decompose) \text{ has not been applied with the same } \mathcal{M} \text{ and } s \text{ before}$$

where $(\mathcal{M}' \Vdash t)^{T \gg \mathcal{M}}$ is $\mathcal{M}' \cup T \Vdash t$ if $\mathcal{M} \subseteq \mathcal{M}'$ and $\mathcal{M}' \Vdash t$ otherwise, $(\cdot)^{T \gg \mathcal{M}}$ extends as expected, $eq(\sigma) = x_1 \doteq t_1 \wedge \ldots \wedge x_n \doteq t_n$ is the constraint

corresponding to a substitution $\sigma = [x_1 \mapsto t_1, \ldots, x_n \mapsto t_n]$, and $mgu(s \doteq t)$ is the standard most general unifier for the pair of terms t and s (in the free-algebra as defined in [MM82]).

Recall that the mgu, if it exists, is unique modulo renaming (mgu extends as expected). Let us now explain the rules. (*Unify*) expresses that one way to generate a term t from knowledge $\mathcal{M}$ is to use any term $s \in \mathcal{M}$ that can be unified with t, but one commits in this case to the unifier σ; this is done by applying σ to the rest of the constraint and recording its equations. (*Unify*) cannot be applied when s or t are variables; intuitively: when t is a variable, the solution is an arbitrary term, so we consider this a solved state (until elsewhere a substitution is required that substitutes t); when s is variable, then it is a subterm of a message that the intruder created earlier. If the earlier constraint is already solved (i.e., a variable) then s is something the intruder could generate from an earlier knowledge and thus redundant.

(*Equation*), which similarly allows us to solve an equation, can be applied if s or t are variables, provided the conditions are satisfied, but later we will have to prevent vacuous application of this rule to its previous result, i.e., the equations $eq(\sigma)$. (*PubConsts*) says that the intruder can generate all public constants.

(*Compose*) expresses that one way to generate a composed term $f(t_1, \ldots, t_n)$ is to generate the subterms $t_1, \ldots, t_n$ (because then f can be applied to them). (*Decompose*) expresses that we can attempt decryption of any term in the intruder knowledge according to the `Ana` function. Recall that Table 7.1 provides examples of `Ana`, and note that for variables or constants Table 7.1 will yield $(\emptyset, \emptyset)$, i.e., there is nothing to analyze. However, if there is a set T of messages that can potentially be obtained if we can derive the keys K, and T is not yet a subset of the knowledge $\mathcal{M}$ anyway, then one way to proceed is to add $\mathcal{M} \Vdash k$ for each $k \in K$ to the constraint store, i.e., committing to finding the keys, and under this assumption we may add T to $\mathcal{M}$ and in fact to any knowledge $\mathcal{M}'$ that is a superset of $\mathcal{M}$. Also for this rule we must prevent vacuous repeated application, such as applying analysis directly to the newly generated $\mathcal{M} \Vdash k$ constraints.

The reduction of constraints deals with conjuncts of the form $\mathcal{M} \Vdash t$ and $s \doteq t$. However, we also have to handle negative substitutions, i.e., conjuncts of the form $\neg\exists\bar{x}.\phi_\sigma$. We show that we can easily check them for satisfiability.

Definition 8.6 A constraint ϕ is *simple*, written simple(ϕ), iff $\phi = \phi_1 \wedge \ldots \wedge \phi_n$ such that for each ϕ_i $(1 \leq i \leq n)$:

- if $\phi_i = \mathcal{M} \Vdash t$, then $t \in \mathcal{V}$;

- if $\phi_i = s \doteq t$, then $s \in \mathcal{V}$ and s does not appear elsewhere in ϕ;
- if $\phi_i = \neg\exists\bar{x}.\phi_\sigma$, then $mgu(\theta(\phi_\sigma)) = \emptyset$ for $\theta = [\bar{y} \mapsto \bar{c}]$ where $\bar{y}$ are the free variables of ϕ_i and $\bar{c}$ fresh constants that do not appear in ϕ.

Theorem 7 *If* simple(ϕ)*, then* ϕ *is satisfiable.*

Proof. From simple(ϕ), by the definition of simple (Definition 8.6), it follows that ϕ is a conjunction of intruder deduction constraints of the form $\mathcal{M} \Vdash x$ with $x \in \mathcal{V}$, equations $x \doteq t$ where $x \in \mathcal{V}$ and where x does not occur elsewhere in ϕ, and inequalities. Let $\bar{y}$ be all variables that occur freely in intruder deduction constraints and inequalities, and let $\theta = [\bar{y} \mapsto \bar{c}]$ for new constants $\bar{c} \in \mathcal{C}_{pub}$ (that do not occur in ϕ and are pairwise different). We show that $\theta(\phi)$ is satisfiable.

All intruder deduction constraints are satisfiable since the constants are in $\mathcal{C}_{pub}$ and the intruder can access those constants by the rule (*Public*) as in Definition 3.2.

The equations are obviously satisfiable: all equations in ϕ have the form $v_i \doteq u_i$, with variables $\bar{v}$ that do not occur elsewhere in ϕ, which implies that $dom(\theta) \cap \bar{v} = \emptyset$, and thus that $\theta(v_i \doteq u_i) = v_i \doteq \theta(u_i)$. All these equations are therefore satisfiable by instantiating every $v_i \in \bar{v}$ with the term u_i.

It remains to show that the inequalities are satisfiable under θ. Let $\phi_0 = \neg\exists\bar{x}.\,\phi_\sigma$ with $\phi_\sigma = \bigwedge s_i \doteq t_i$ be any inequality. $\theta(\phi_0)$ is closed, i.e., $fv(\theta(\phi_0)) = \emptyset$. This implies that $fv(\phi_\sigma) = \{\bar{x}\}$, and since ϕ is simple, we have $mgu(\theta(\phi_\sigma)) = \emptyset$. Then, ϕ_σ is not satisfiable, i.e., there do not exist $\bar{x}$ such that ϕ_σ holds. Thus, ϕ_0 holds. □

The completeness of the symbolic intruder constraint reduction is similar to existing results on symbolic intruder constraints; what is particular is our generalization to constraints with quantified inequalities. To that end, we first show:

Lemma 1 *Let* $\phi = \neg\exists\bar{x}.\phi_\sigma$ *where* $\phi_\sigma = \bigwedge s_i \doteq t_i$*, and let* $\theta = [\bar{y} \mapsto \bar{c}]$ *where* $\bar{y} = fv(\phi)$ *and* $\bar{c}$ *are fresh public constants that do not occur in* ϕ*. Then* ϕ *is satisfiable iff* $\theta(\phi)$ *is satisfiable. Moreover,* ϕ *is satisfiable iff* $mgu(\theta(\phi_\sigma)) = \emptyset$*.*

Proof. If ϕ is unsatisfiable, then also $\theta(\phi)$ is unsatisfiable. For the other

direction, we show that the following two formulas are in contradiction:

$$\exists \bar{y}. \forall \bar{x}. \bigvee_{i=1}^{n} s_i \neq t_i \tag{8.1}$$

$$\exists \bar{x}. \bigwedge_{i=1}^{n} \theta(s_i) = \theta(t_i) \tag{8.2}$$

By (8.2), we can find a substitution $\xi = [\bar{x} \mapsto \bar{u}]$ where $\bar{u}$ are ground terms such that $\bigwedge_{i=1}^{n} \xi(\theta(s_i)) = \xi(\theta(t_i))$. Since θ and ξ are substitutions with disjoint domain and grounding, we have $\theta(\xi(\cdot)) = \xi(\theta(\cdot))$, and thus we obtain

$$\bigwedge_{i=1}^{n} \theta(\xi(s_i)) = \theta(\xi(t_i)) \tag{8.3}$$

By (8.1), choosing a particular value for the $\bar{x}$, we obtain:

$$\exists \bar{y}. \bigvee_{i=1}^{n} \xi(s_i) \neq \xi(t_i) \tag{8.4}$$

Then we can find an $i \in \{1, ..., n\}$ such that $\exists \bar{y}. \xi(s_i) \neq \xi(t_i)$. Thus, taking $s := \xi(s_i)$ and $t := \xi(t_i)$, we have:

$$\exists \bar{y}. s \neq t \tag{8.5}$$

$$\theta(s) = \theta(t) \tag{8.6}$$

To show that (8.5) and (8.6) yield a contradiction, we consider all possible cases of s and t:

- If s and t are atomic, then, since θ replaces all of variables $\bar{y}$ with fresh constants, $\theta(s) = \theta(t)$ implies $s = t$, contradicting (8.5).
- If s is atomic and t is not, then, since $\theta(s)$ is a constant, $\theta(s) \neq \theta(t)$, contradicting (8.6).
- If both s and t are not atomic, then $s = f(s_1, \ldots, s_n)$ and $t = f(t_1, \ldots, t_n)$ (otherwise $\theta(s) = \theta(t)$ cannot hold). Thus, we can reduce this case to one pair s_i and t_i of corresponding subterms.

Now, since $\theta(\phi)$ is closed, i.e., $fv(\phi_\sigma) = \bar{x}$, we can decide the satisfiability of ϕ with the *mgu*-algorithm. □

Now we prove the soundness, completeness and termination of the satisfiability calculus of the symbolic intruder, but let us first define what we mean by

soundness and completeness. For that let us write $\phi \to \phi'$ if $\frac{\phi'}{\phi}$ is an instance of a reduction rule, i.e., representing one solution step. By *sound* we mean that for a model $\mathcal{I}$, $\mathcal{I} \models \phi'$ and $\phi \to \phi'$ imply $\mathcal{I} \models \phi$. Moreover, by *complete* we mean that for a model $\mathcal{I}$ and a non-simple ϕ, if $\mathcal{I} \models \phi$ then exists a ϕ' such that $\phi \to \phi'$ and $\mathcal{I} \models \phi'$.

Theorem 8 (Adaption of [RT03, Möd12a]) *The satisfiability calculus for the symbolic intruder is* sound, complete, *and terminating on well-formed constraints.*

Proof.

Soundness is straightforward since for each rule $\frac{\phi'}{\phi}$, from a satisfying interpretation of an instance $\sigma(\phi')$ of ϕ', we can derive an interpretation that satisfies $\sigma(\phi)$.

Completeness is more complicated, i.e., when $\mathcal{I} \models \phi$, then either ϕ is already simple or we can apply some rule, obtaining $\phi \to \phi'$ for some ϕ' with $\mathcal{I} \models \phi'$. Thus, we show that every model $\mathcal{I}$ of a constraint is preserved by at least one applicable reduction rule until we obtain a simple constraint (that we already know is satisfiable by Theorem 7). Consider a satisfiable non-simple constraint ϕ, and a satisfying interpretation $\mathcal{I}$. Since $\mathcal{I}$ satisfies ϕ, for every intruder deduction $\mathcal{M} \Vdash t$ in ϕ, there exists a proof $\mathcal{I}(\mathcal{M}) \Vdash \mathcal{I}(t)$ using the intruder deduction rules of Definition 7.1. This proof has a tree shape with $\mathcal{I}(\mathcal{M}) \Vdash \mathcal{I}(t)$ at the root and axioms as leaves for members of $\mathcal{I}(\mathcal{M})$. We label each $\mathcal{M} \Vdash t$ with such a proof for $\mathcal{I}(\mathcal{M}) \Vdash \mathcal{I}(t)$.

We now proceed from the first (in the order induced by the well-formedness of ϕ) intruder constraint $\mathcal{M} \Vdash t$ where $t \notin \mathcal{V}$ (i.e., not yet simple) and show: depending on the form of the derivation tree, we can pick a rule so that we can label all new deduction constraints in the resulting constraint ϕ' again with matching proof trees, i.e., so that they support still the solution. In particular, we will apply the (*Unify*) rule only with substitutions of which $\mathcal{I}$ is an instance.

If ϕ contains a non-simple equation, then we can apply the (*Equation*) rule to simplify it, because ϕ is satisfiable under $\mathcal{I}$. Thus, $\mathcal{I}(s) = \mathcal{I}(t)$ and so there is a $\sigma \in mgu(s \doteq t)$, with $\mathcal{I}(x) = \mathcal{I}(\sigma(x))$ for all $x \in \mathcal{V}$. Therefore, the resulting constraint (replacing $s \doteq t$ by $eq(\sigma)$ and applying σ to the rest of the constraint) still has $\mathcal{I}$ as a model.

If all equations are simple, then for ϕ to be non-simple, there must be at least one conjunct $\mathcal{M}_i \Vdash t_i$ where $t_i \notin \mathcal{V}$. Consider the smallest such i (in the order

of the well-formedness of ϕ, thus $fv(\mathcal{M}_i) \subseteq \{t_1, \ldots, t_{i-1}\} \subseteq \mathcal{V}$). Moreover, consider the ground derivation of $\mathcal{I}(\mathcal{M}_i) \Vdash \mathcal{I}(t_i)$, which exists because ϕ is satisfiable. We distinguish the different cases at the root of this proof tree:

- If it is a leaf, then $\mathcal{I}(t_i) \in \mathcal{I}(\mathcal{M}_i)$, thus t_i has a unifier with some term $s \in \mathcal{M}_i$. Now, t_i cannot be a variable because otherwise this conjunct would be already simple). If s is a variable, then $s = t_j$ for some $j < i$, and we can thus proceed by following the proof tree of t_j instead. If neither t_i nor s are variables, then the (*Unify*) rule is applicable, and again the unifier σ supports $\mathcal{I}$, and so does the resulting constraint.
- If it is an application of the (*Public*) rule, then $t_i \in \mathcal{C}_{pub}$ and so the public constant rule of the constraint reduction is applicable.
- If it is an application of the (*Compose*) rule, then so is the corresponding rule of the constraint reduction, producing a new conjunction $\mathcal{M} \Vdash t'_1 \wedge \ldots \wedge \mathcal{M} \Vdash t'_l$ of deduction constraints for the immediate subterm t'_j of t_i; we can label these t_j with the respective subtrees of the derivation tree of t_i, so the resulting constraint still supports the interpretation $\mathcal{I}$.
- If the node is an application of the (*Decompose*) rule, then consider the ground term t that is being decomposed in the derivation proof for $\mathcal{I}(t_i)$. We first consider different cases depending on how t is derived:
  - If it is a composition step, then the intruder composed the term and then decomposed it subsequently. Since decomposition can only yield subterms of the composed term, one of the subtrees proves that the intruder can already derive $\mathcal{I}(t_i)$ and we can thus simplify the proof tree. We thus assume in the following that the proof tree contains no composition followed by a decomposition.
  - It cannot be an application of the (*Public*) rule, since that cannot have an analyzable subterm.
  - If t is obtained by a decomposition step itself, then we regress to the respective term being decomposed, and we do so until we hit a term that is not obtained by decomposition. By the previous cases, this cannot be a composition step or public-constant step either, so all remains is following case:
  - The derivation of t is a leaf, i.e., there is a $t' \in \mathcal{M}$ such that $\mathcal{I}(t') = t$. We now show that in this case we can perform the decomposition step.

  Since decomposition is performed on t in the derivation of $\mathcal{I}(t_i)$, we have that $\texttt{Ana}(t) = (K, T)$ for some sets of ground terms K and T, where $\mathcal{I}(\mathcal{M}) \Vdash k$ for every $k \in K$ and $\mathcal{I}(t_i) \in T$.

We have two further cases, namely whether t' (the term in $\mathcal{M}$ whose instance is t) is a variable or not. If t' is a variable, then again $t' = t_j$ for some $j < i$ and we can just replace the subtree for the derivation of $\mathcal{I}(t')$ with the derivation of $\mathcal{I}(t_j)$.

Finally, if t' is not a variable, then $\texttt{Ana}(t') = (K', T')$ for some sets K' and T' with $\mathcal{I}(K') = K$ and $\mathcal{I}(P') = T$. Unless $T' \subseteq \mathcal{M}$ (which is for instance the case if the decomposition step has already been applied previously, so we can simply replace the decomposition step with a leaf node), we can apply the decomposition rule of the constraint reduction and label the newly added conjuncts $\mathcal{M}_i \Vdash k'$ for every $k' \in K'$ with the respective derivation of $\mathcal{I}(k')$ of the previous proof tree. Thus, the resulting constraint (also extending all $\mathcal{M}_j$ that are supersets of $\mathcal{M}_i$ with the terms from P') supports the interpretation $\mathcal{I}$.

For termination, it is standard to define a *weight* (n, m, l) for a constraint ϕ, where

- n is the number of free variables in ϕ;
- m is the number of unanalyzed subterms in the intruder knowledges of constraints, i.e., let $\mathcal{M}_i^*$ be the set of all terms in $\mathcal{M}_i$ and their subterms to which analysis has not yet been applied, and let $m = \Sigma_i |\mathcal{M}_i^*|$, where $|\mathcal{M}_i^*|$ is the number of unanalyzed terms in the set $\mathcal{M}_i^*$ and $\Sigma_i |\mathcal{M}_i^*|$ is the sum for all $\mathcal{M}_i$ in a constraint;
- $l = \mathit{size}(\phi)$, where

$$\begin{aligned}
&\mathit{size}(\phi \wedge \phi') = \mathit{size}(\phi) + \mathit{size}(\phi')\\
&\mathit{size}(\mathcal{M} \Vdash t) = \mathit{size}(t)\\
&\mathit{size}(s \doteq t) = \mathit{size}(s) + \mathit{size}(t)\\
&\mathit{size}(c) = \mathit{size}(x) = 1\\
&\mathit{size}(\neg\exists.\phi_\sigma) = 0\\
&\mathit{size}(f(t_1, \ldots, t_n)) = \mathit{size}(t_1) + \ldots + \mathit{size}(t_n) + 1
\end{aligned}$$

We order the components of this weight lexicographically, i.e., $(n, m, l) > (n', m', l')$ iff $n > n'$ or $(n = n'$ and $(m > m'$ or $(m = m'$ and $l > l')))$. Obviously, $>$ has no infinite descending chain. Now, for every derivation step $\phi \to \phi'$

- either ϕ' has a smaller number of variables than ϕ ((*Unify*) or (*Equation*) with a substitution $\sigma \neq \mathit{identity}$), thus the n component is smaller,
- or we apply (*Decompose*), marking the analyzed term (also in supersets of the respective knowledge) and thus decrease the m component, note that

the side condition $T \not\subseteq \mathcal{M}$ of this rule prevents us from getting infinite cycles when applying the rule,

- or we apply *Unify* or *Equation* with a substitution $\sigma = \mathit{identity}$, or we apply any of the other rules, therefore in all these cases the l component decreases.

So, every $\rightarrow$ step reduces the weight and termination then follows quite straightforwardly. □

8.5 Summary

In this chapter, we defined a protocol as a set of operational strands and a set of goals expressed in the geometric fragment. Accordingly, we defined the semantics of a protocol as an infinite-state transition system over symbolic states. Then we showed how to translate any goal in the geometric fragment for a given symbolic state $(\mathcal{S}; \mathcal{M}; E; \phi)$ into a constraint ϕ', so that the models of $\phi \wedge \phi'$ represent exactly all concrete instances of the state $(\mathcal{S}; \mathcal{M}; E; \phi)$ that violate the goal. To solve such constraints, we defined the satisfiability calculus for the symbolic intruder as a rule-based procedure, and proved its correctness in light of [RT03, Möd12a]. This procedure provides the basis for our typing and parallel composition results presented in the next chapter.

Chapter 9

Typing and Compositionality Results

In this chapter present our relative soundness results. First, we define our typed model and give the typing result in Section 9.1. Then, in Section 9.2, we define the parallel composition of protocols and give the compositionality result. In Section 9.3 we present the *Automatic Protocol Composition Checker (APCC)*.

9.1 Typed Model

In our typed model, the *set of all possible types for terms* is denoted by $\mathcal{T}_{\hat{\Sigma}\cup\mathfrak{T}_a}$, where $\mathfrak{T}_a$ is a finite set of *atomic types*, e.g., $\mathfrak{T}_a = \{\texttt{Number}, \texttt{Agent}, \texttt{PublicKey}, \texttt{PrivateKey}, \texttt{SymmetricKey}\}$. We call all other types *composed types*. Each atomic term (each element of $\mathcal{V} \cup \mathcal{C}$) is given a type; constants are given an *atomic type* and variables are given either an atomic or a composed type (any element of $\mathcal{T}_{\hat{\Sigma}\cup\mathfrak{T}_a}$). We write $t : \tau$ to denote that a term t has the type τ. Based on the type information of atomic terms, we define the *typing function* Γ as follows:

Definition 9.1 Given $\Gamma(\cdot) : \mathcal{V} \to \mathcal{T}_{\hat{\Sigma}\cup\mathfrak{T}_a}$ for variables and $\Gamma(\cdot) : \mathcal{C} \to \mathfrak{T}_a$ for constants, we extend Γ to map all terms to a type, i.e., $\Gamma(\cdot) : \mathcal{T}_{\hat{\Sigma}\cup\mathcal{C}}(\mathcal{V}) \to \mathcal{T}_{\hat{\Sigma}\cup\mathfrak{T}_a}$,

as follows: $\Gamma(t) = f(\Gamma(t_1), \cdots, \Gamma(t_n))$ if $t = f(t_1, \cdots, t_n)$ and $f \in \hat{\Sigma}^n$. We say that a substitution σ is *well-typed* iff $\Gamma(x) = \Gamma(\sigma(x))$ for all $x \in dom(\sigma)$.

For example, if $\Gamma(k) = \texttt{PrivateKey}$ and $\Gamma(x) = \texttt{Number}$ then $\Gamma(\texttt{crypt}(\texttt{pub}(k), x)) = \texttt{crypt}(\texttt{pub}(\texttt{PrivateKey}), \texttt{Number})$.

As we require that all constants be typed, we further partition $\mathcal{C}$ into disjoint countable subsets according to different types in $\mathfrak{T}_a$. This models the intruder's ability to access infinite reservoirs of public fresh constants. For example, for protocols P_1, P_2 and $\mathfrak{T}_a = \{\beta_1, \ldots, \beta_n\}$, we have the disjoint subsets $\mathcal{C}_{pub}^{\beta_i}$, $\mathcal{C}_{priv}^{\beta_i}$, $\mathcal{C}_{P_1}^{\beta_i}$ and $\mathcal{C}_{P_2}^{\beta_i}$, where $i \in \{1, \ldots, n\}$ and, e.g., $\mathcal{C}_{pub}^{\beta_i}$ contains all $\mathcal{C}_{pub}$ elements of type β_i. $\mathcal{C}_{P_1}^{\beta_i}$ and $\mathcal{C}_{P_2}^{\beta_i}$ are short-term constants, whereas $\mathcal{C}_{pub}^{\beta_i}$ and $\mathcal{C}_{priv}^{\beta_i}$ are long-term, and we consider it an attack if the intruder learns any of $\mathcal{C}_{priv}^{\beta_i}$.

By an easy induction on the structure of terms, we have:

Lemma 2 *If a substitution σ is well-typed, then $\Gamma(t) = \Gamma(\sigma(t))$ for all terms $t \in \mathcal{T}_{\hat{\Sigma} \cup \mathcal{C}}(\mathcal{V})$.*

Proof. By induction on the structure of the term t.
Base case: $t \in V$. Since $\Gamma(t)$ is defined then for a well-typed σ; if $t \in dom(\sigma)$ then $\Gamma(t) = \Gamma(\sigma(t))$ (by definition of well-typed substitution). Otherwise the lemma holds trivially as $t = \sigma(t)$ and thus $\Gamma(t) = \Gamma(\sigma(t))$.
Induction Step: Given a well-typed σ and the terms $t_1, \ldots, t_n$ such that for each t_i it holds that $\Gamma(t_i) = \Gamma(\sigma(t_i))$. For $t = f(t_1, \ldots, t_n)$, by definition of Γ follows that $\Gamma(\sigma(t)) = f(\Gamma(\sigma(t_1)), \cdots, \Gamma(\sigma(t_n))) = f(\Gamma(t_1), \cdots, \Gamma(t_n)) = \Gamma(\sigma(t)) = \Gamma(t)$. □

According to this typed model, $\mathcal{I}$ is a *well-typed interpretation* iff $\Gamma(x) = \Gamma(\mathcal{I}(x))$ for all $x \in \mathcal{V}$. Moreover, we require for the typed model that $\Gamma(s) = \Gamma(t)$ for each $s \doteq t$. This is a restriction only on the strands of the honest agents (as they are supposed to act honestly), not on the intruder: he can send ill-typed messages freely. We later show that sending ill-typed messages does not help the intruder in introducing new attacks in protocols that satisfy certain conditions.

Message Patterns

In order to prevent the intruder from using messages of a protocol to attack a second protocol, we need to guarantee the disjointness of the messages between

both protocols. Thus, we use formats to wrap raw data, as discussed in Chapter 7.1. In particular, all submessages of all operators (except formats and public key operator) must be "wrapped" with a format, e.g., $\mathtt{scrypt}(k, \mathtt{f}_a(Na))$ and $\mathtt{scrypt}(k, \mathtt{f}_b(Nb))$ should be used instead of $\mathtt{scrypt}(k, Na)$ and $\mathtt{scrypt}(k_1, Nb)$.

We define the set of protocol message patterns, where we need to ensure that each pair of terms has disjoint variables: we thus define a well-typed α-renaming $\alpha(t)$ that replaces the variables in t with completely new variable names.

Definition 9.2 The *message pattern* of a message t is $MP_t(t) = \{\alpha(t)\}$. Moreover, the *set* $MP_S(S)$ *of message patterns of a strand* S is as follows:

$$\begin{array}{lcl}
MP_S(\mathsf{send}(t).S) & = & MP_t(t) \cup MP_S(S) \\
MP_S(\mathsf{event}(t).S) & = & MP_t(t) \cup MP_S(S) \\
MP_S(\mathsf{receive}(t).S) & = & MP_t(t) \cup MP_S(S) \\
MP_S(s \doteq t.S) & = & MP_t(s) \cup MP_t(t) \cup MP_S(S)
\end{array}$$

$$\begin{array}{lcl}
MP_S((\exists \bar{x}.\phi_\sigma).S) & = & MP_S(\phi_\sigma) \cup MP_S(S) \\
MP_S((\neg\exists \bar{x}.\phi_\sigma).S) & = & MP_S(\phi_\sigma) \cup MP_S(S) \\
MP_S(0) & = & \emptyset
\end{array}$$

Finally, the *set* $MP_\Psi(\Psi)$ *of message patterns of a goal* Ψ is defined as follows:

$$\begin{array}{lcl}
MP_\Psi(\forall x.\psi \Rightarrow \psi_0) & = & MP_\Psi(\psi) \cup MP_\Psi(\psi_0) \\
MP_\Psi(\mathsf{ik}(t)) & = & MP_t(t) \\
MP_\Psi(\mathsf{event}(t)) & = & MP_t(t) \\
MP_\Psi(\psi_1 \vee \psi_2) & = & MP_\Psi(\psi_1) \cup MP_\Psi(\psi_2) \\
MP_\Psi(\psi_1 \wedge \psi_2) & = & MP_\Psi(\psi_1) \cup MP_\Psi(\psi_2) \\
MP_\Psi(s \doteq t) & = & MP_t(s) \cup MP_t(t) \\
MP_\Psi(\neg\phi) & = & MP_t(\phi)
\end{array}$$

The *set of message patterns of a protocol* $P = (\{S_1, \cdots, S_m\}; \{\Psi_1, \cdots, \Psi_n\})$ is $MP(P) = \bigcup_{i=1}^{m} MP_S(S_i) \cup \bigcup_{i=1}^{n} MP_\Psi(\Psi_i)$, and the *set of sub-message patterns of a protocol* P is $SMP(P) = \{\alpha(s) \mid t \in MP(P) \wedge s \sqsubseteq t \wedge \neg atomic(s)\} \setminus \{u \mid u = pub(v)$ for some term $v\}$. *SMP* applies to messages, strands, goals as expected.

Example 9.1 *If* $S = \mathsf{receive}(\mathtt{scrypt}(k, (\mathtt{f}_1(x, y)))).\mathsf{send}(\mathtt{scrypt}(k, y))$*, then* $SMP(S) = \{\mathtt{scrypt}(k, \mathtt{f}_1(x_1, y_1)), \mathtt{scrypt}(k, y_2), \mathtt{f}_1(x_3, y_3)\}$. □

Definition 9.3 A protocol $P = (\mathcal{S}_0, G)$ is *type-flaw-resistant* iff the following conditions hold:

- $MP(P)$ and $\mathcal{V}$ are disjoint, i.e., $MP(P) \cap \mathcal{V} = \emptyset$ (which ensures that none of the messages of P is sent as raw data). Note that this condition is without loss of generality; i.e., wrapping data with formats does not change the security of a protocol; because the intruder (and all other agents) can compose and decompose these wrapped messages (formats are transparent public functions).

- If two non-atomic sub-terms are unifiable, then they have the same type, i.e., for all $t_1, t_2 \in SMP(P)$, if $\sigma(t_1) = \sigma(t_2)$ for some σ, then $\Gamma(t_1) = \Gamma(t_2)$.

- For any equation $s \doteq t$ that occurs in strands or goals of P (also under a negation), $\Gamma(s) = \Gamma(t)$.

- For any variable x that occurs in equations or events of G, $\Gamma(x) \in \mathfrak{T}_a$.

- For any variable x that occurs in inequalities or events of strands, $\Gamma(x) \in \mathfrak{T}_a$.

These conditions (of Definition 9.3) are a sort of sanity check on the specification of protocols and, although they provide a slight restriction on the expressiveness, they enable us to prove (in Theorem 9) that even if the intruder sends ill-typed messages, he will not cause a new attack on a type-flaw-resistant protocol. Moreover, even with these conditions, we can still express many goals with the geometric fragment that go beyond secrecy and authentication, and consider a wide class of protocols (see Chapter 9.3). The first condition ensures that the protocol messages are wrapped using formats and not sent as raw data. Note that this a realistic modeling and not a restriction on the intruder ability, i.e., formats are a sound abstraction that does not eliminate attacks. The second condition means that we cannot unify two terms unless their types match. Note that this match is a restriction on honest agents only, and the intruder is still able to send ill-typed messages.

Example 9.2 *Example 9.1 had a potential type-flaw vulnerability as* $\mathtt{scrypt}(k, \mathtt{f}_1(x_1, y_1))$ *and* $\mathtt{scrypt}(k, y_2)$ *have the unifier* $[y_2 \mapsto \mathtt{f}_1(x_1, y_1)]$. *Here* y_1 *and* y_2 *must have the same type since they have been obtained by a well-typed variable renaming in the construction of SMP. Thus, the two messages have different types. The problem is that the second message encrypts raw data without any information on who it is meant for and it may thus be mistaken for the first message. Let us thus change the second message to* $\mathtt{scrypt}(k, \mathtt{f}_2(y_2))$. *Then SMP also includes* $\mathtt{f}_2(y_4)$ *for a further variable* y_4, *and now no two different elements of SMP have a unifier.* $\mathtt{f}_2$ *is not necessarily inserting a tag: if the type of* y *in the implementation is a fixed-length type, this is already sufficient for distinction.* □

Lemma 3 *Let ϕ be a simple constraint where $\Gamma(s) = \Gamma(t)$ holds for all equations $s \doteq t$, and whether the equation is under a negation (i.e., part of an inequality) then neither s nor t contain variables of composed types. Then, ϕ has a well-typed model, i.e., a well-typed interpretation $\mathcal{I}^T$ with $\mathcal{I}^T \models \phi$.*

Proof. For this proof, we first show that we can find a well-typed model for all inequalities (with values in $\mathcal{C}_{pub}$ that the intruder can generate). Consider an inequality $\phi = \neg\exists\bar{x}.\phi_\sigma$. Consider also a substitution θ of all free variables of ϕ (which are of atomic types) with constants of $\mathcal{C}_{pub}$ with corresponding types. Since ϕ is simple, the equations of $\theta(\phi_\sigma)$ cannot have a unifier. We have thus found a well-typed model for all the free variables of ϕ_σ. It is straightforward to extend this to a well-typed model of the entire constraint; since all positive equations must be well-typed, and the intruder has an access to infinite reservoirs of fresh constants of all atomic types. □

Theorem 9 *If a type-flaw-resistant protocol P has an attack, then P has a well-typed attack.*

Proof. The key idea is to consider a satisfiable constraint $\Phi = \phi \wedge tr_{\mathcal{M},E}(\Psi)$ that represents an attack against P, i.e., where ϕ is the constraint of a reachable state of P and $tr_{\mathcal{M},E}(\Psi)$ is the translation of the violated goal Ψ in that state. We have to show that the constraint has also a well-typed solution. By Theorem 8 and since Φ is satisfiable, we can use the symbolic intruder reduction rules to obtain a simple constraint Φ', i.e., $\Phi \rightarrow^* \Phi'$. The point is now that for a type-flaw-resistant protocol, all substitutions in this reduction are well-typed. To prove this we have to show that if $P = (\mathcal{S}_0, \{\Psi_0, \cdots, \Psi_n\})$ is a type-flaw-resistant protocol and there exist an attack state $(\mathcal{S}; \mathcal{M}; E; \phi)$ such that $(\mathcal{S}_0; \emptyset; \emptyset, \top) \Rightarrow^* (\mathcal{S}; \mathcal{M}; E, \phi)$ and an interpretation $\mathcal{I}$ such that $\mathcal{I}, \mathcal{M}, E \models_{\mathfrak{S}} \neg\Psi_i$ and $\mathcal{I} \models \phi$, then there exists a well-typed interpretation $\mathcal{I}^T$ such that $\mathcal{I}^T, \mathcal{M}, E \models_{\mathfrak{S}} \neg\Psi_i$ and $\mathcal{I}^T \models \phi$. Recall that, by Theorem 6, $\mathcal{I}, \mathcal{M}, E \models_{\mathfrak{S}} \neg\Psi_i$ iff $\mathcal{I} \models tr_{\mathcal{M},E}(\Psi_i)$, and thus $\mathcal{I}^T, \mathcal{M}, E \models_{\mathfrak{S}} \neg\Psi_i$ iff $\mathcal{I}^T \models tr_{\mathcal{M},E}(\Psi_i)$.

Note that ϕ is initially $\top$, which is well-typed by default (Definition 9.3), and since P is a type-flaw-resistant protocol, we have that: (1) all equations and events in the initial set of strands $\mathcal{S}_0$ are well-typed, (2) the type of each variable in the inequalities of the strands is atomic, and (3) $\Rightarrow$ does not introduce any negations (so no equality becomes an inequality or vice versa). Hence, all equations in ϕ are well-typed, and variables in inequalities have atomic types.

Recall that $\Psi_i = \forall\bar{x}.\psi \Longrightarrow \psi_0$. We show now that $tr_{\mathcal{M},E}$ does not change these properties that originally hold in Ψ_i, i.e., $tr(\Psi_i)$ has the same properties that Ψ_i has by the definition of type-flaw-resistant protocol. This means that $tr_{\mathcal{M},E}$ does not preserves the well-typedness.

We show this by cases on $tr'_{\mathcal{M},E}$, since $tr(\cdot)$ passes Ψ_i to $tr'_{\mathcal{M},E}(\cdot)$ with a negation on the ψ_0; that is fine since all equations (positive and negative) in Ψ_i must be well-typed, and all variables that occur in such equations must have atomic types. Now we prove by induction that $tr'_{\mathcal{M},E}$ preserves the above properties:

- $tr'_{\mathcal{M},E}(\mathsf{event}(t)) = \bigvee_{\mathsf{event}(s)\in E} s \doteq t$: all events in E originate from the initial set of strands $\mathcal{S}_0$, and since P is a type-flaw-resistant protocol, then all equation $s \doteq t$ that we derive from these events (when applying $tr_{\mathcal{M},E}$ on Ψ_i) are well-typed.
- $tr'_{\mathcal{M},E}(\neg\mathsf{event}(t)) = \bigwedge_{\mathsf{event}(s)\in E} \neg s \doteq t$: we can conclude reasoning similarly to the previous cases where we already discussed the negation in $tr_{\mathcal{M},E}$.
- $tr'_{\mathcal{M},E}(s \doteq t) = s \doteq t$: immediate.
- $tr'_{\mathcal{M},E}(\exists \bar{x}.\psi) = \exists \bar{x}.tr'_{\mathcal{M},E}(\psi)$: immediate.
- $tr'_{\mathcal{M},E}(\neg s \doteq t) = \neg s \doteq t$: immediate.
- $tr'_{\mathcal{M},E}(\neg\exists \bar{x}.\, \mathsf{event}(t_1) \wedge \cdots \wedge \mathsf{event}(t_n) \wedge u_1 \doteq v_1 \wedge \cdots \wedge u_m \doteq v_m) = \bigwedge_{\mathsf{event}(s_1)\in E \ldots \mathsf{event}(s_n)\in E} \neg\exists \bar{x}.(s_1 \doteq t_1 \wedge \cdots \wedge t_n \doteq s_n \wedge u_1 \doteq v_1 \wedge \cdots \wedge u_m \doteq v_m)$: follows by reduction to previous cases.
- $tr'_{\mathcal{M},E}(\psi_1 \vee \psi_2) = tr'_{\mathcal{M},E}(\psi_1) \vee tr'_{\mathcal{M},E}(\psi_2)$ and the rest of the cases: follow by reduction to previous cases.

We need to prove that for a type-flaw-resistant protocol, if there is an interpretation $\mathcal{I} \models \Phi$, then there is a well-typed $\mathcal{I}^T \models \Phi$. By Theorem 8 and since Φ is satisfiable, we can use the symbolic intruder reduction rules to obtain a simple constraint Φ', i.e., $\Phi \rightarrow^* \Phi'$ that supports $\mathcal{I}$.

Next we prove that all substitutions made in the reduction steps from Φ to Φ' are well-typed substitutions. We prove this as follows.

Let us define SMP_0 to be the closure of SMP under (1) subterm relation, (2) well-typed α-renaming, and (3) unification, i.e., if two terms of SMP_0 are unifiable then the corresponding instances are also in SMP_0. Note that by definition of type-flaw-resistant protocols (Definition 9.3), all equations $s \doteq t$ (including inequalities) of Φ have $\Gamma(s) = \Gamma(t)$, and for inequalities, we have that all variables of s and t are of atomic type. Further, all terms that occur are well-typed instances of terms in SMP_0. Moreover, for a type-flaw resistant protocol, all substitutions in this reduction must be well-typed.

To prove this, we first consider the two cases that involve substitutions in the constraint reduction rules, namely *(Unify)* and *(Equation)*. For the *(Unify)* rule, we proceed by cases of s and t:

- If both s and t are atomic, then s and t cannot be variables, so the above property is preserved trivially, simply because they must be the same constant.
- If both are composed, then $\sigma(s) = \sigma(t)$ and there exist $s, t \in SMP_0$. Then, $\sigma(s) = \sigma(t)$ and thus $\Gamma(s) = \Gamma(t)$ as the protocol is type-flaw-resistant, and so σ is well-typed.

For the *(Equation)* rule, we conclude immediately from the fact that P is type-flaw-resistant; since all equations are well-typed, all unifications must be well-typed.

In case of the other rules (that generate subterms) the properties are preserved immediately; since first these rules do not incorporate any substitution, and the generated subterms of these rules are in SMP_0.

So far we have arrived at a simple(Φ') where $\Gamma(s) = \Gamma(t)$ holds for all equations $s \doteq t$, and if the equation is under a negation (i.e., part of an inequality), then neither s nor t contain variables of composed types. By Lemma 3, we conclude the existence of well-typed solution for such Φ'. □

Note that this theorem does not exclude that type-flaw attacks are possible, but rather says that for every type-flaw attack there is also a (similar) well-typed attack, so it is safe to verify the protocol only in the typed model.

9.2 Parallel Composition

In this section, we consider the parallel composition of protocols, in the following we will often speak of just "composition". We define the set of operational strands for the composition of a pair of protocols as the union of the sets of the operational strands of the two protocols; this allows all possible transitions in the composition. The goals for the composition are also the union of the goals of the pair, since any attack on any of them is an attack on the whole composition (i.e., the composition must achieve the goals of the pair).

Definition 9.4 The *parallel composition* $P_1 \parallel P_2$ of $P_1 = (\mathcal{S}_0^{P_1}; \Psi_0^{P_1})$ and $P_2 = (\mathcal{S}_0^{P_2}; \Psi_0^{P_2})$ is $P_1 \parallel P_2 = (\mathcal{S}_0^{P_1} \cup \mathcal{S}_0^{P_2};\ \Psi_0^{P_1} \cup \Psi_0^{P_2})$.

Our parallel composition result relies on the following key idea. Similar to the typing result, we look at the constraints produced by an attack trace against $P_1 \parallel P_2$, violating a goal of P_1, and show that we can obtain an attack against P_1 alone, or a leaking of a long-term secret by P_2 alone. Again, the core of this proof is the observation that the unification steps of the symbolic intruder never produce an "ill-typed" substitution in the sense that a P_1-variable is never instantiated with a P_2 message and vice versa. For that to work, we have a similar condition as before, namely that the non-atomic subterms of the two protocols (the SMPs) are disjoint, i.e., each non-atomic message uniquely says to which protocol it belongs. This is more liberal than the requirements in previous parallel compositionality results in that we do not require a particular tagging scheme: any way to make the protocol messages distinguishable is allowed. Further, we carefully set up the use of constants in the protocol as explained in Section 9.1, namely that all constants used in the protocol are either: long-term public values that the intruder initially knows; long-term secret values that, if the intruder obtains them, count as a secrecy violation in *both* protocols; or short-term values of P_1 or of P_2.

A major technical difficulty for this construction is the relationship between public and private keys. For instance, note that in this thesis, public keys are not an atomic type, but rather we use terms of the form $\mathsf{pub}(t)$ where t is a private key. There are of course other ways to model this, e.g., using a (private) function from public to private keys, but it is impossible to avoid that either public or private keys are non-atomic terms[1]. This in turn leads to the problem that in a constraint reduction we could run for instance into the following situation: the intruder has learned a public key $\mathsf{pub}(x)$ from protocol P_1 and should use it to produce a public key $\mathsf{pub}(y)$ in protocol P_2 (using the unify rule). Such an ill-typed unification would destroy the compositionality argument. There are several ways to avoid this:

- Forbid that the two protocols share asymmetric keys, i.e., only one protocol may use the pub operator. This would render the approach quite unusable as there are few modern protocols that use only symmetric key cryptography.

- Require that the argument of pub can only be a constant, meaning that in all asymmetric encryptions and signatures, the key has to be fixed. This however prevents us from modeling protocols where an agent learns

[1] There are several papers that consider an "extra-logical" function $\cdot^{-1}$ that maps from constants of type public key to constants of type private key; extra-logical means that this function is not part of $\hat{\Sigma}$ and could thus not be used in terms directly, because it would be unclear how to define x^{-1} for a variable x, and thus this solution does not work with symbolic approaches.

a public key $\mathsf{pub}(x)$ from a message (e.g., a certificate) and then uses it. In fact this is what some previous parallel composition results do [cCC10].

Our setup of constants allows however for a third way that is far less restrictive to protocols: we require that all constants of type private key are either part of the long-term secrets or long-term public constants. (The latter is all private keys that belong to the intruder.) Moreover, the intruder can obtain all public keys, i.e., $\mathsf{pub}(c)$ for every c of type private key. This does not prevent the honest agents from creating fresh key-pairs (the private key shall be chosen from the long-term constants as well) but it dictates that each private key is either a perpetual secret (it will be an attack if the intruder obtains it) or it is public right from the start (as all public keys are). This only excludes protocols where a private key is a secret at first and later revealed to the intruder, or where some public keys are initially kept secret.

In a nutshell, the only limitation of our model is that long-term secrets cannot be "declassified": we require that all constants of type private key are either part of the long-term secrets or long-term public constants.

Definition 9.5 Two protocols P_1 and P_2 are *parallel-composable* iff the following conditions hold:

(1) P_1 and P_2 are *SMP-disjoint*, i.e., for every $s \in SMP(P_1)$ and $t \in SMP(P_2)$, either s and t have no unifier ($mgu(s \doteq t) = \emptyset$) or $s = \mathsf{pub}(s_0)$ and $t = \mathsf{pub}(t_0)$ for some s_0, t_0 of type private key.

(2) All constants of type private key that occur in $MP(P_1) \cup MP(P_2)$ are part of the long-term constants in $\mathcal{C}_{pub} \cup \mathcal{C}_{priv}$.

(3) All constants that occur in $MP(P_i)$ are in $\mathcal{C}_{pub} \cup \mathcal{C}_{priv} \cup \mathcal{C}_{P_i}$, i.e., are either long term or belong to the short-term constants of the respective protocol.

(4) For every $c \in \mathcal{C}_{P_i}^{\mathtt{PrivateKey}}$, P_i also contains the strand $\mathsf{send}(\mathsf{pub}(c)).0$.

(5) For each secret constant $c \in \mathcal{C}_{priv}^{\beta_i}$, for each type β_i, each P_i contains the strands $\mathsf{event}(lts_{\beta_i,P_i}(c)).0$ and the goal $\forall x : \beta_i.\, \mathsf{ik}(x) \implies \neg lts_{\beta_i,P_i}(x)$.

(6) Both P_1 and P_2 are type-flaw resistant.

Some remarks on the conditions:

- Condition (1) is the core of the compositionality result, as it helps to avoid confusion between messages of the two protocols;

- Condition (2) ensures that every private key is either initially known to the intruder or is part of the long-term secrets (and thus prevents "declassification" of private keys as we discussed above).
- Condition (3) means that the two protocols will draw from disjoint sets of constants for their short-term values.
- Condition (4) ensures that public keys are known to the intruder. Note that typically the goals on long-term secrets, like private keys and shared symmetric keys, are very easy to prove as they are normally not transmitted. The fact that we do not put all public keys into the knowledge of the intruder in the initial state is because the intruder knowledge must be a finite set of terms for the constraint reduction to work. Putting it into strands means they are available at any time, but the intruder knowledge in every reachable state (and thus constraint) is finite. Similarly, for the goals on long-term secrets: the set of events in every reachable state is still finite, but for every leaked secret, we can in one transition reach the corresponding predicate that triggers a violation of the secrecy goal.
- Condition (5) ensures that when either protocol P_i leaks any constant of $\mathcal{C}_{priv}^{\beta_i}$, it is a violation of its secrecy goals.
- Condition (6) ensures that for both protocols, we cannot unify terms unless their types match.

Theorem 10 *If two protocols P_1 and P_2 are parallel-composable and both P_1 and P_2 are secure in isolation in the typed model, then $P_1 \parallel P_2$ is secure (also in the untyped model).*

Proof. Consider an attack against $P_1 \parallel P_2$ violating a goal Ψ of P_1. We show that the given attack works also on P_1 in isolation, or one of the P_i in isolation leaks one of the long-term secret constants. We use a similar argument as in Theorem 9: let ϕ be a constraint that represents the attack against $P_1 \parallel P_2$; we show how to extract a satisfiable constraint that represents either an attack against P_1 or P_2 in isolation, and that this attack works in the typed model.

First observe that composability of P_1 and P_2 implies that they are both type-flaw resistant. Thus, there is no unifier between two messages of $SMP(P_1)$ unless they have the same type, and the same holds for $SMP(P_2)$. Since also there are no unifiers between a message from $SMP(P_1)$ and a message from $SMP(P_2)$, we can derive that also $P_1 \parallel P_2$ is type-flaw resistant. By Theorem 9, if there is an attack against $P_1 \parallel P_2$, then there must be also a well-typed attack against $P_1 \parallel P_2$. We can thus without loss of generality assume that the given constraints that represent an attack against $P_1 \parallel P_2$ have a well-typed solution.

This allows us first to get rid of all variables of composed types: we substitute each variable of a composed type $f(\tau_1, \ldots, \tau_n)$ by the expression $f(x_1, \ldots, x_n)$ where the x_i are new variables of types τ_i, and repeat this process until we have only variables of atomic types. Since the given constraint has a well-typed solution, so has the transformed one. Let $\mathcal{I}$ be such a well-typed solution.

As a second step, we substitute every variable x of type private key with $\mathcal{I}(x)$.[2] Thus we have no variables of type private key anymore, and for every $\mathsf{pub}(t)$, t is a public or secret long-term constant of type private key.

The next step in the proof is to introduce a label for each term and subterm that occurs in a $\mathcal{M} \Vdash t$ constraint in ϕ, namely whether the corresponding term "belongs" to P_1 or to P_2. Recall that in $\mathcal{M} \Vdash t$, t represents a message that the intruder sent to an honest agent (or was required to construct for violating a goal), and $\mathcal{M}$ represents messages he has received from honest agents. By construction, all these terms are thus instances of either an $MP_S(P_1)$ or an $MP_S(P_2)$ term and can be labeled as such, with the exception of $\mathsf{pub}(t)$ terms in an intruder knowledge $\mathcal{M}$ which would be available in both. Let us thus label all occurrences of $\mathsf{pub}(t)$ with label $\star$. For each term, we give the same label to all its subterms. Note that throughout the constraint, all occurrences of a variable thus receives the same labeling and we can thus speak of P_1-variables and P_2-variables. Similarly, all fresh constants $\mathcal{C}_{P_1}$ will be labeled P_1 and all fresh constants of $\mathcal{C}_{P_2}$ will be labeled P_2. However the public and private long-term constants may occur both with a P_1 and with a P_2 label; let us thus label them with $\star$ instead. In this way, all terms and their subterms have a consistent labeling in the sense that all occurrences of the term will bear the same label. Also by construction, in all equations $s \doteq t$, both s and t are labeled P_1 or both P_2.

As shown already in the proof of Theorem 9, during constraint reduction we obtain only terms that are instances of $SMP(P_1)$ or $SMP(P_2)$ or that are atomic. What we now show is that for one of the P_i all those constraints $\mathcal{M} \Vdash t$ where t is labeled P_i can be solved using only messages in $\mathcal{M}$ that are also labeled P_i or $\star$ (and are not a long-term secret constant). This means that the constraint $\mathcal{M} \Vdash t$ can still be solved when removing from $\mathcal{M}$ all messages labeled for the other protocol, so that we have an attack that works in the respective P_i alone.

To that end, as in the proof of Theorem 9, we proceed along the well-formedness order of the constraint, solving the first non-simple one in a way that supports the given solution $\mathcal{I}$, and show that during this constraint reduction we never

[2] The reader may wonder why we would not do this actually for all variables. In fact, we cannot, because in general the solution $\mathcal{I}$ may be exploiting that messages from P_1 can be used for P_2. Only for private keys, we know that they are either long-term secrets or long-term public values, either (supposedly) secret in both protocols or available in both.

need to use a P_2 message to solve a P_1 constraint or vice versa. In particular, we will never perform a unification between a P_1-labeled and a P_2-labeled message.

Whenever applying the (*Equation*) rule on an equation $s \doteq t$, it is impossible that s is labeled P_1 and t is labeled P_2 (or vice versa) by construction, so also the resulting unifier produces equations with this property.

More critical is the (*Unify*) rule, solving $\mathcal{M} \Vdash t$ by unifying t with some term $s \in \mathcal{M}$. Suppose t is labeled P_1 or $\star$; s may be labeled P_1, P_2 or $\star$. We show that there is a solution without using a P_2 message. Since (*Unify*) requires that $s, t \notin \mathcal{V}$, they either are both the same constant or they are both composed. We distinguish the following cases:

- If both are the same constant c, then we have any of the following subcases:
  - $c \in \mathcal{C}_{pub}$: then the constraint can rather be solved using the (*PubConsts*) rule instead (and thus there is no need to use any message from P_2).
  - $c \in \mathcal{C}_{priv}$: then $\mathcal{M}$ contains a long-term secret constant. Let P_i be the protocol from which c was learned. Since all previous constraints (in the well-formedness order) are already simple, those that are labeled P_i form a valid trace of P_i alone that leaks c and thus can be extended to an attack against P_i (disregarding all further constraints).
  - $c \in \mathcal{C}_{P_1}$ or $c \in \mathcal{C}_{P_2}$, then by construction they are both labeled P_1 or both labeled P_2, so (*Unify*) does not destroy our invariant.
- If both are composed, then we further distinguish two cases:
  - $s = t = \mathsf{pub}(c)$: we have that $\mathsf{pub}(c)$ is available in both protocols by parallel-composability, i.e., in order to solve this in a P_1 constraint without using P_2, we can augment the attack trace with an initial step where the intruder learns $\mathsf{pub}(c)$ from the respective strand of P_1.
  - In all other cases, we can use again that all non-atomic messages are instances of terms in $SMP(P_1)$ or $SMP(P_2)$ and that the protocols are SMP-disjoint, so if s and t have a unifier, they must belong to the same $SMP(P_i)$.

Finally, consider analysis step: in this case, when analyzing a message $m \in \mathcal{M}$, we obtain constraints of the form $\mathcal{M} \Vdash k$ for some subterm k of m. Moreover, we add then some subterm p of m to $\mathcal{M}$ in the analyzed constraint. Here, k and p must have the same label as m or be labeled $\star$. It follows that to analyze

a P_1-labeled message, we will never need to construct a P_2-labeled key (or vice versa).

In conclusion, we thus never need P_1 messages to solve a P_2 constraint or vice versa, with the exception of leaked long-term secret constants. In this case, however, we have already found an attack in an initial part of the constraint (in which all P_1 constraints can be solved without P_2 and vice versa). Thus, we obtain in all cases a sequence of constraints for the individual protocols, and one of them is an attack. □

We can then apply this theorem successively to any number of protocols that satisfy the conditions, in order to prove that they are all parallel composable.

This compositionality result entails an interesting observation about parallel composition with insecure protocols: unless one of the protocols leaks a long-term secret, the intruder never needs to use one protocol to attack another protocol. This means actually: even if a protocol is flawed, it does not endanger the security of the other protocols as long as it at least manages not to leak the long-term secrets. For instance, the Needham-Schroeder Public Key protocol has a well-known attack, but the intruder can never obtain the private keys of any honest agent. Thus, another protocol relying on the same public-key infrastructure is completely unaffected. This is a crucial point because it permits us to even allow for security statements in presence of flawed protocols:

Corollary 9.6 *Consider two protocols P_1 and P_2 that are parallel-composable (and thus satisfy all the conditions in Definition 9.5). If P_1 is secure in isolation and P_2, even though it may have an attack in isolation, does not leak a long-term secret, then all goals of P_1 hold also in $P_1 \parallel P_2$.*

9.3 Automated Protocol Composition Checker

The *Automated Protocol Composition Checker* APCC [3], was developed as a part of the SPS compiler. APCC implements the check for the two main syntactic conditions of our results: it checks both whether a given protocol is type-flaw-resistant and whether the protocols in a given set are pairwise parallel composable. Figure 9.1 shows the architecture of APCC including its input and output.

[3] Available at http://www2.compute.dtu.dk/~samo/SPS.zip.
The main implementation effort was carried out by Dr. Paolo Modesti, our partner in the FutureID project

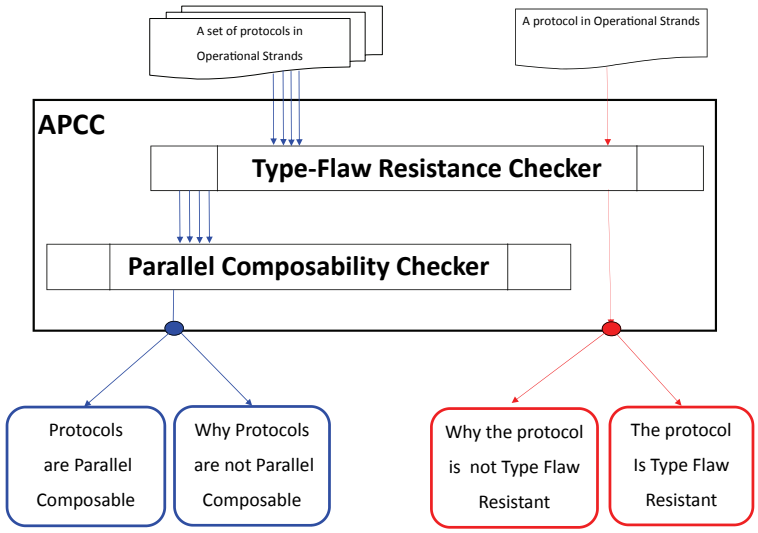


Figure 9.1: The APCC tool

In our preliminary experiments, we considered a suite that includes widely used protocols like TLS, Kerberos (PKINIT and Basic) and protocols defined by the ISO/IEC 9798 standard, along with well-known academic protocols (variants of Needham-Schroeder-Lowe, Denning-Sacco, etc.). Although we worked with abstract and simplified models, we were able to verify that TLS and Kerberos are parallel composable. In contrast, since some protocols of the ISO/IEC 9798 standard share common formats, they are not *SMP*-disjoint.

For many academic protocols, our tool give the result that they are not pairwise parallel composable. This result was expected because these protocols do not have a standardized implementation, and thus the format of messages at the wire level is not part of the specification. In fact, in these protocols there are several terms that may be confused with terms of other protocols, whereas a concrete implementation may avoid this by choosing carefully disjoint messages formats that can prevent the unification. Hence, our tool APCC can also support developers in the integration of new protocols (or new implementations of them) in an existing system.

Finally, for the entire test suite, the type-flaw resistance test took less than half a second, while the parallel-composability test took less than 0.2 of a second on

a 2.67 GHz machine.

9.4 Summary

We presented our relative soundness results for typing and compositional reasoning. We proved that a type-flaw-resistant protocol admits an attack if it admits an attack in the typed model. We also proved that — for a pair of parallel-composable protocols — their parallel composition will not introduce new attacks, i.e., if a parallel composition of protocol admits an attack, then one of the protocols in isolation admits an attack. This result is of course applicable to a set of protocols by applying it successively to any number of protocols that satisfy the conditions. Based on our results, the APCC tool was developed to check if a protocol or a set of protocols satisfies our conditions.

Chapter 10

Related Work

This work unifies research on the soundness of typed models (e.g., [HLS03, BP05, Möd12a, AD14]) and on parallel protocol composition (e.g., [GT00, Gut09, CD09, cCC10, ACG+08]) by using a proof technique that has been employed in both areas: attack reduction based on a symbolic constraint systems. For typing, the idea is that the constraint solving never needs to apply ill-typed substitutions if the protocol satisfies some sufficient conditions; hence, for every attack there exists a well-typed variant and it is thus without loss of generality to restrict the model to well-typed execution. For the parallel composition of P_1 and P_2 that again satisfy some sufficient conditions, the constraint solving never needs to use a message that the intruder learned from P_1 to construct a message of P_2; thus, the attack will work in P_1 alone or in P_2 alone, and from verifying them in isolation, we can conclude that their composition is secure.

We also make several generalizations over previous results. First, the result of [AD07] is limited to encryptions, signing and pairing; while we allow a variety of operators like the predefined `hash` and `mac` functions, and the user defined functions, this widens the class of protocols for which our results are applicable. User defined functions also include formats that allow for a more realistic typing and subsumes pairing and tagging that they use. Note that they have a finer grained typing system compared to ours; as they have a distinct data type for each nonce, fresh key and constant, e.g., in Needham-Schroeder protocol, each of the two nonces has its own data type. However, our results are applicable

to finer grained typing systems with no cost. Second, our work is not limited to atomic keys, this limitation excluded many interesting protocols like TLS that use composed keys. Instead, we allow such composed keys by means of not only the pre-defined hash function, but also by the use of any other user defined function. Third, we are not limited to a fixed PKI, instead we allow the creation of public/private key pairs. (This limitation is also present in the works of [AD14, CD09].) This leads us to our next generalization point that we make over [AD07], that is we start from a completely untyped model. However, they start from what we may call a "partially" typed model, i.e., the types of keys and agents (that they refer as principals) must be maintained from the starting model and all principals must keep these types. We also allow the occurrence of blind copies unlike the aforementioned result. (A blind copy occurs when an agent sends a part of a message that he received without knowing what this part contains). Last, we are not limited to a fixed set of properties like secrecy. Instead, we consider the entire geometric fragment proposed by Guttman [Gut14]. In fact this is a generalization over other works like [BP05]. One could characterize this fragment as allowing a "controlled" amount of negation in the goal specifications. We believe is the most expressive language that can work with the given constraint-solving argument that is at the core of handling typing and compositionality results uniformly.

Other expressive property languages have been considered, e.g., *PS*-LTL for typing results [AD14]; an in-depth comparison of the various existing property languages and their relative expressiveness is yet outstanding. Although equalities are not supported in *PS*-LTL, the temporal operators and arbitrary negation in it allows the specification of some properties like fairness that we cannot support in our fragment.

Moreover, early works on typing and parallel composition used a fixed tagging scheme, whereas we use the more general notion of non-unifiable subterms for messages that have different meaning. Using the notion of formats, our results are applicable to existing real-world protocols like TLS with their actual formats. Moreover, our definitions of type-flaw resistant protocols and parallel-composable protocols can act as a set of design principles (such as disjointness of the message format when messages have a different meaning) avoid many problems by construction already as in [AN96, Sha08].

Our work considered so far protocols only in the initial term algebra without any algebraic properties. There are some promising results for such properties (e.g., [KT09, cCC10, Möd11]) that we would like to combine with our approach in the future. The same holds for other types of protocol composition, e.g., the sequential composition considered in [cCC10], where one protocol establishes a key that is used by another protocol as input.

Part III

Conclusion

Chapter 11

Contributions and Future Work

In this thesis, we explore some aspects of the specification, implementation, verification and composition of security protocols. We defined the (SPS) Security Protocol Specification language that not only offers several modeling features suitable for real-world protocols like TLS and EAC, but also enjoys a formal semantics that is mathematically simpler than any previous attempt. For a fixed set of operators — that enables the modeling of a wide range of real-world protocols — we implemented a compiler that translates SPS specifications to operational strands upon which we defined the semantics of SPS language. From operational strands and within the SPS compiler, we developed automatic translators to robust real-world implementations and corresponding formal models. We have demonstrated practical feasibility with a number of major and minor case studies, including TLS and the EAC/PACE protocols used in the German eID card. Another component of SPS compiler is the Automatic Protocol Composition Checker, that implements our relative soundness results for protocol typing and compositionality. The typing result shows that if a type-flaw-resistant protocol has an attack, then it has a well-typed attack. For protocol compositionality, we consider the parallel composition of protocols, and we show that if running two protocols in parallel admits an attack, then at least one of them has an attack in isolation; given that those protocols are parallel composable.

Now we summarize the contributions of this thesis.

11.1 Contributions

The main contributions of this thesis are as follows:

Specification, Implementation and Verification of Security Protocols

- We give a precise and concise semantics for Alice-and-Bob style languages in the light of [Möd09, CR10], which we further simplify considerably. Our semantics depends on few definitions and parameterized over arbitrary algebraic theories of the cryptographic operators, and precisely define how message derivation and checking are performed by a translation to operational strands. This translation enables further connections to formal models and implementations, and achieves independence of the approach of the verification tools or the implementation language. While many underlying questions of the translation are undecidable in this generality, we prove that our translator correctly implements the semantics for a fixed theory. This set of operators is representative for what is commonly used, and for which it is known how to soundly encode the algebraic reasoning into tools like ProVerif.
- In addition to the fixed set of operators that enables the modeling of a wide range of real-world protocols, SPS gives the users the ability to add more operators to model different aspects of protocols. Users can define operators in the form of: formats, free functions, and mappings.
  - A format is a one-to-one connection between various mechanisms of structuring messages in implementations and abstract constructors in formal models. Formats replace the abstract concatenation operator from formal models allowing us to generate code for any real-world structuring mechanism like XML formats or TLS-style messages. We systematically use the non-crypto API that offers a Java class for each format that basically implements a parser and pretty-printer for that format (i.e., serializer and de-serializer in protocol implementation slang). We shift thus all problems of type-flaw attacks, insertion attacks, and buffer-overflows that actual implementations can have to the implementation of the respective Java class. In fact, it is then easy to devise translators from format specification languages (like XML schema) to an implementation of these Java classes.

  - Free-functions represent standard one way functions that all agents can use to compose messages.
  - Mappings associate different protocol objects to each other and represent bonds that exist in reality but not in the form of functions, e.g., a shared key between two agents A and B is simply modeled using the mapping $\texttt{shk}(A, B)$ that does not exist in reality. Based on mappings, we handle a "classical" problem of formal protocol models that is the instantiation of the roles with honest and dishonest agents in an unbounded number of sessions, as well as the relation between honest agents and their long term keys. We generate the instantiation in a systematic way that follows a simple principle: for any number of sessions, every role can be played by any agent, including the intruder.

- Together with allowing users to define their own operators, SPS allows users to customize data types and cryptographic primitives. Since actual protocols require parameterization like key-lengths, group parameters, and the concrete crypto algorithms/variants; We allow here either that an SPS specification uses one fixed configuration, or that one annotates the operators in the specification. This means that users can annotate types and cryptographic operators with attributes in the form of name/value pairs, e.g., $\texttt{A} \rightarrow \texttt{B} : \texttt{crypt}[alg = RSA, keysize = 2048](k, m)$ that says `A` sends to `B` the message m encrypted with the public key k and that the algorithm is *RSA* with *keysize* of *2048* bits. The annotated details are transparent to the generated implementation. The parameters can also be transmitted in the messages of the protocol (and may be subject of authentication goals for instance).

Typing and Compositionality of Security Protocols

- We develop a set of design principles, for example, messages of different meanings should have disjoint formats. Following such engineering practices will avoid many problems by construction already.

- Based on the definition of the design principles, we unify and simplify existing typing and compositionality results: we recast them as an instance of the same basic principle and of the same proof technique. In a nutshell, this technique is to reduce the search for attacks to constraint solving in a symbolic model. For protocols that satisfy the respective sufficient conditions, constraint reduction will never make an ill-typed substitution, while for compositionality "ill-typed" means to unify messages from two different protocols.

- This systematic approach also allows us to significantly generalize existing results to a larger set of protocols and security properties. For what concerns protocols, our soundness results do not require a particular fixed tagging scheme like most previous works, but use more liberal requirements that are satisfied by many existing real-world protocols like TLS. While many existing results are limited to simple secrecy goals, we prove our results for the entire geometric fragment suggested by Guttman [Gut14]. We even augment this fragment with the ability to directly refer to the intruder knowledge in the antecedent of goals; while this does not increase expressiveness, it is very convenient in specifications. In fact, handling the geometric fragment also constitutes a slight generalization of existing constraint-reduction approaches.
- For practical applicability, the APCC tool was developed to check if protocols meet the design principles, i.e., satisfy the sufficient conditions of type-flaw resistance and parallel-composability.

11.2 Future Work

The future directions that seem to be promising and deserve further investigation include:

- Widening the class of protocols that SPS can specify; i.e., improving the expressiveness of SPS. This involves:
  - Expressing goals in the geometric fragment instead of using the built-in secrecy and authentication goals.
  - Embedding control structures such as deterministic selection, non-deterministic selection, and sub-protocol calls.
  - Including database definition and operators like add, delete and query a database.
  - Extending our protocol suite that involves a deeper study for the actual formats and theory of new protocols and encoding new protocols like JPAKE [HR10].
- The improvement of SPS expressiveness facilitates the connecting of SPS to other state of the art tools like Tamarin [MSCB13], Set-π [BMNN15], SAPIC [KK14], or AIF [Möd10]. Those tools enable the handling of protocols with loops, non-monotonic global states, and databases, and therefore such connections would definitely widen the spectrum of protocols that SPS can handle.

- The improvements may also go beyond the symbolic model like linking SPS to tools in the computational model like CryptoVerif [Bla07].
- An interesting future work inspired by the work of [BM09] is the verification of our results using Isabelle [NPW02].
- Another promising venue for future work would be specifying privacy notation in SPS based on $\alpha\beta$ notion of privacy as defined in [MGV13], and define privacy goals on this formalization. That is translating what we may call $\alpha\beta$-SPS to an $\alpha\beta$ transition system and check that all β leak nothing more than α. This can form a basis for a front-end to translate static equivalence questions to a tool like Yapa [BCD13].
- As our work in typing and compositional reasoning currently considered protocols only in the free-algebra: the initial term algebra without any algebraic properties, there are some promising results for such properties (e.g., [KT09, cCC10, Möd11]) that we plan to investigate and thus integrate into our approach.
- Another promising direction involves the consideration for other types of protocol composition, e.g., (1) the sequential composition considered in [cCC10], where one protocol establishes a key that is used by another protocol as input, (2) the vertical composition of protocols as in [GM11, MV14] where an application protocol runs over a channel established by another protocol.

Bibliography

[AAA+12] Alessandro Armando, Wihem Arsac, Tigran Avanesov, Michele Barletta, Alberto Calvi, Alessandro Cappai, Roberto Carbone, Yannick Chevalier, Luca Compagna, Jorge Cuéllar, Gabriel Erzse, Simone Frau, Marius Minea, Sebastian Mödersheim, David von Oheimb, Giancarlo Pellegrino, SerenaElisa Ponta, Marco Rocchetto, Michael Rusinowitch, Mohammad Torabi Dashti, Mathieu Turuani, and Luca Viganò. The AVANTSSAR Platform for the Automated Validation of Trust and Security of Service-Oriented Architectures. In *TACAS*, LNCS 7214, pages 267–282. Springer, 2012.

[AC04] A. Armando and L. Compagna. SATMC: A SAT-based Model Checker for Security Protocols. *Logics in Artificial Intelligence*, pages 730–733, 2004.

[AC06] Martín Abadi and Véronique Cortier. Deciding knowledge in security protocols under equational theories. *Theoretical Computer Science*, 367(1-2):2–32, 2006.

[ACC+08] Alessandro Armando, Roberto Carbone, Luca Compagna, Jorge Cuéllar, and M. Llanos Tobarra. Formal analysis of SAML 2.0 web browser single sign-on: breaking the saml-based single sign-on for google apps. In *Proceedings of the 6th ACM Workshop on Formal Methods in Security Engineering, FMSE 2008, Alexandria, VA, USA, October 27, 2008*, pages 1–10, 2008.

[ACG+08] Suzana Andova, Cas J. F. Cremers, Kristian Gjøsteen, Sjouke Mauw, Stig Fr. Mjølsnes, and Sasa Radomirovic. A framework

for compositional verification of security protocols. *Inf. Comput.*, 206(2-4):425–459, 2008.

[AD07] Myrto Arapinis and Marie Duflot. Bounding messages for free in security protocols. In *FSTTCS 2007: Foundations of Software Technology and Theoretical Computer Science, 27th International Conference, New Delhi, India, December 12-14, 2007, Proceedings*, pages 376–387, 2007.

[AD14] Myrto Arapinis and Marie Duflot. Bounding messages for free in security protocols - extension to various security properties. *Inf. Comput.*, 239:182–215, 2014.

[AF01] Martín Abadi and Cédric Fournet. Mobile values, new names, and secure communication. In Chris Hankin and Dave Schmidt, editors, *POPL*, pages 104–115. ACM, 2001.

[AMMV15a] Omar Almousa, Sebastian Mödersheim, Paolo Modesti, and Luca Viganò. Typing and compositionality for security protocols: A generalization to the geometric fragment. In *Computer Security - ESORICS 2015 - 20th European Symposium on Research in Computer Security, Vienna, Austria, September 21-25, 2015, Proceedings, Part II*, pages 209–229, 2015.

[AMMV15b] Omar Almousa, Sebastian Mödersheim, Paolo Modesti, and Luca Viganò. Typing and Compositionality for Security Protocols: a Generalization to the Geometric Fragment (Extended Version), 2015. Available at http://www.imm.dtu.dk/~samo/.

[AMV15a] Omar Almousa, Sebastian Mödersheim, and Luca Viganò. Alice and Bob: Reconciling Formal Models and Implementation, 2015. Programming languages with applications to biology and security - Colloquium in honour of Pierpaolo Degano for his 65th birthday, Revised Selected and Invited Papers.

[AMV15b] Omar Almousa, Sebastian Mödersheim, and Luca Viganò. Alice and Bob: Reconciling Formal Models and Implementation (Extended Version). Technical report, DTU Compute, 2015. Available at http://www.imm.dtu.dk/~samo/.

[AN96] Martín Abadi and Roger M. Needham. Prudent engineering practice for cryptographic protocols. *IEEE Trans. Software Eng.*, 22(1):6–15, 1996.

[BBD+05] Chiara Bodei, Mikael Buchholtz, Pierpaolo Degano, Flemming Nielson, and Hanne Riis Nielson. Static validation of security protocols. *Journal of Computer Security*, 13(3):347–390, 2005.

[BBH12] Michael Backes, Alex Busenius, and Catalin Hritcu. On the Development and Formalization of an Extensible Code Generator for Real Life Security Protocols. In *NFM*, LNCS 7226, pages 371–387. Springer, 2012.

[BCD13] Mathieu Baudet, Véronique Cortier, and Stéphanie Delaune. YAPA: A generic tool for computing intruder knowledge. *ACM Trans. Comput. Log.*, 14(1):4, 2013.

[BFCZ12] Karthikeyan Bhargavan, Cédric Fournet, Ricardo Corin, and Eugen Zalinescu. Verified Cryptographic Implementations for TLS. *ACM Trans. Inf. Syst. Secur.*, 15, 2012.

[BFGT06] Karthikeyan Bhargavan, Cédric Fournet, Andrew D. Gordon, and Stephen Tse. Verified interoperable implementations of security protocols. In *19th Computer Security Foundations Workshop, (CSF '19 2006), 5-7 July 2006, Venice, Italy*, pages 139–152. IEEE, 2006.

[BKRS15] David Basin, Michel Keller, Sasă Radomirović, and Ralf Sasse. Alice and Bob Meet Equational Theories. In *Festschrift for José Meseguer on his 65th Birthday*, 2015.

[Bla01] Bruno Blanchet. An efficient cryptographic protocol verifier based on Prolog rules. In *IEEE 14th Computer Security Foundations Symposium (CSF '01)*, pages 82–96. IEEE, 2001.

[Bla07] Bruno Blanchet. Cryptoverif: Computationally sound mechanized prover for cryptographic protocols. In *Dagstuhl seminar "Formal Protocol Verification Applied*, page 117, 2007.

[BM09] Achim D. Brucker and Sebastian Mödersheim. Integrating automated and interactive protocol verification. In *Formal Aspects in Security and Trust, 6th International Workshop, FAST 2009, Eindhoven, The Netherlands, November 5-6, 2009, Revised Selected Papers*, pages 248–262, 2009.

[BMNN15] Alessandro Bruni, Sebastian Mödersheim, Flemming Nielson, and Hanne Riis Nielson. Set-Pi: Set Membership p-Calculus. In *IEEE The 28th Computer Security Foundations Symposium (CSF '15)*, pages 185–198, 2015.

[BN98] Franz Baader and Tobias Nipkow. *Term rewriting and all that.* Cambridge University Press, 1998.

[BN07] Sébastien Briais and Uwe Nestmann. A formal semantics for protocol narrations. *Theoretical Computer Science*, 389(3):484–511, 2007.

[BP05] Bruno Blanchet and Andreas Podelski. Verification of cryptographic protocols: tagging enforces termination. *Theor. Comput. Sci.*, 333(1-2):67–90, 2005.

[BS11] Bruno Blanchet and Ben Smyth. ProVerif 1.85: Automatic Cryptographic Protocol Verifier, User Manual and Tutorial, 2011.

[BSC14] Bruno Blanchet, Ben Smyth, and Vincent Cheval. Proverif 1.90: Automatic cryptographic protocol verifier, user manual and tutorial, 2014.

[Car94] U. Carlsen. Generating formal cryptographic protocol specifications. In *IEEE Symposium on Research in Security and Privacy*, pages 137—146, 1994.

[cCC10] Ştefan Ciobâcă and Véronique Cortier. Protocol composition for arbitrary primitives. In *IEEE 23rd Computer Security Foundations Symposium (CSF' 10)*, pages 322–336. IEEE, 2010.

[CD09] Véronique Cortier and Stéphanie Delaune. Safely composing security protocols. *Form. Method. Syst. Des.*, 34:1–36, 2009.

[CDM11] Hubert Comon-Lundh, Stéphanie Delaune, and Jonathan K. Millen. Constraint solving techniques and enriching the model with equational theories. In *Formal Models and Techniques for Analyzing Security Protocols*, pages 35–61. IOS Press, 2011.

[CEvdGP87] David Chaum, Jan-Hendrik Evertse, Jeroen van de Graaf, and René Peralta. Demonstrating possession of a discrete logarithm without revealing it. In *Advances in Cryptology—CRYPTO'86*, pages 200–212. Springer, 1987.

[CHY07] Marco Carbone, Kohei Honda, and Nabuko Yoshida. Structured global programming for communication behaviour. In *Proceedings of the 16th European Symposium on Programming Languages and Systems (ESOP)*, volume 4421, pages 2–17, 2007.

[CJ95] John A. Clark and Jeremy Jacob. On the security of recent protocols. *Inf. Process. Lett.*, 56(3):151–155, 1995.

[CJ97] John Clark and Jeremy Jacob. A survey of authentication protocol literature: Version 1.0, 1997.

[CJM00] Edmund M Clarke, Somesh Jha, and Will Marrero. Verifying security protocols with brutus. *ACM Transactions on Software Engineering and Methodology (TOSEM)*, 9(4):443–487, 2000.

[CKR+03] Yannick Chevalier, Ralf Küsters, Michaël Rusinowitch, Mathieu Turuani, and Laurent Vigneron. Extending the dolev-yao intruder for analyzing an unbounded number of sessions. In *Computer Science Logic, 17th International Workshop, CSL 2003, 12th Annual Conference of the EACSL, and 8th Kurt Gödel Colloquium, KGC 2003, Vienna, Austria, August 25-30, 2003, Proceedings*, pages 128–141, 2003.

[CM05] Cas Cremers and Sjouke Mauw. Operational semantics of security protocols. In *Scenarios: Models, Transformations and Tools*, volume 3466, pages 66–89, 2005.

[CMR01] Véronique Cortier, Jonathan Millen, and Harald Rueß. Proving secrecy is easy enough. In *14th IEEE Computer Security Foundations Workshop*. IEEE Computer Society, 2001.

[CR10] Yannick Chevalier and Michaël Rusinowitch. Compiling and securing cryptographic protocols. *Information Processing Letters*, 110(3):116–122, 2010.

[Cre06] Casimier J. F. Cremers. *Scyther: Semantics and verification of security protocols*. Eindhoven University of Technology, 2006.

[CVB06] Carlos Caleiro, Luca Viganò, and David Basin. On the semantics of Alice&Bob specifications of security protocols. *Theoretical Computer Science*, 367(1):88–122, 2006.

[DR08] Tim Dierks and Eric Rescorla. RFC 5246: The Transport Layer Security (TLS) Protocol, Version 1.2, 2008.

[Eas11] Donald Eastlake. RFC6066: Transport Layer Security (TLS) Extensions: Extension Definitions, 2011.

[EMM09] Santiago Escobar, Catherine Meadows, and José Meseguer. Maude-NPA: Cryptographic protocol analysis modulo equational properties. *Foundations of Security Analysis and Design V*, pages 1–50, 2009.

[Fed12] Federal Office for Information Security. Advanced Security Mechanism for Machine Readable Travel Documents - Extended Access Control (EAC), Password Authenticated Connection Establishment (PACE), and Restricted Identification (RI). Technical Guideline BSI-TR-03110, Version 2.10, Part 1 - 3, 2012. https://www.bsi.bund.de/EN/Publications/TechnicalGuidelines/TR03110/BSITR03110.html.

[Fut] The FutureID project. www.futureid.eu.

[Fut13a] FutureID Project. Deliverable D32.1: Survey and Analysis of Existing eID and Credential Systems , 2013.

[Fut13b] FutureID Project. Deliverable D42.2: Interface and Module Specification and Documentation, 2013.

[Fut13c] FutureID Project. Deliverable D42.3: Future AnB: The projected APS Language of FutureID, 2013.

[Fut14] FutureID Project. Deliverable D42.6: Specification of execution environment, 2014.

[Fut15] FutureID Project. Deliverable D42.8: APS Files for Selected Authentication Protocols, 2015.

[Ger08] German Federal Office for Information Security (BSI). Advanced Security Mechanism for Machine Readable Travel Documents, 2008. Available at www.bsi.bund.de/EN/Publications/TechnicalGuidelines/TR03110/BSITR03110.

[GM11] T. Groß and S. Mödersheim. Vertical protocol composition. In *IEEE 24th Computer Security Foundations Symposium (CSF '11)*, pages 235 –250, june 2011.

[GT00] Joshua D. Guttman and F. Javier Thayer. Protocol independence through disjoint encryption. In *IEEE 13th Computer Security Foundations Symposium (CSF '00)*, pages 24–34. IEEE, 2000.

[GTC+04] Joshua D. Guttman, F. Javier Thayer, Jay A. Carlson, Jonathan C. Herzog, John D. Ramsdell, and Brian T. Sniffen. Trust management in strand spaces: A rely-guarantee method. In *Programming Languages and Systems, 13th European Symposium on Programming, ESOP 2004, Held as Part of the Joint European Conferences on Theory and Practice of Software, ETAPS 2004, Barcelona, Spain, March 29 - April 2, 2004, Proceedings*, pages 325–339, 2004.

[Gut09] Joshua D. Guttman. Cryptographic Protocol Composition via the Authentication Tests. In *FOSSACS'09*, pages 303–317. Springer, 2009.

[Gut14] Joshua D. Guttman. Establishing and preserving protocol security goals. *Journal of Computer Security*, 22(2):203–267, 2014.

[HLS03] James Heather, Gavin Lowe, and Steve Schneider. How to prevent type flaw attacks on security protocols. *Journal of Computer Security*, 11(2):217–244, 2003.

[HR10] Feng Hao and Peter Ryan. J-PAKE: authenticated key exchange without PKI. *Transactions on Computational Science*, 11:192–206, 2010.

[Int99] International Organization for Standardization (ISO). ISO/IEC 9798-4:1999. Information technology – Security techniques – Entity authentication – Part 4: Mechanisms using a cryptographic check function, 1999.

[Int08] International Organization for Standardization (ISO). ISO/IEC 9798-2:2008. Information technology – Security techniques – Entity authentication – Part 2: Mechanisms using symmetric encipherment algorithms, 2008.

[Int09] International Organization for Standardization (ISO). ISO/IEC 9798-5:2009. Information technology – Security techniques – Entity authentication – Part 5: Mechanisms using zero-knowledge techniques, 2009.

[Int10a] International Organization for Standardization (ISO). ISO/IEC 9798-1:2010. Information technology – Security techniques – Entity authentication – Part 1: General, 2010.

[Int10b] International Organization for Standardization (ISO). ISO/IEC 9798-3:1998/Amd 1:2010. Information technology – Security techniques – Entity authentication – Part 3: Mechanisms using digital signature techniques, 2010.

[Int10c] International Organization for Standardization (ISO). ISO/IEC 9798-6:2010. Information technology – Security techniques – Entity authentication – Part 6: Mechanisms using manual data transfer, 2010.

[JRV00] Florent Jacquemard, Michaël Rusinowitch, and Laurent Vigneron. Compiling and verifying security protocols. In *Logic for Programming and Automated Reasoning*, LNAI 1955, pages 131–160. Springer, 2000.

[KK14] Steve Kremer and Robert Künnemann. Automated analysis of security protocols with global state. In *2014 IEEE Symposium on Security and Privacy, SP 2014, Berkeley, CA, USA, May 18-21, 2014*, pages 163–178, 2014.

[KT09] Ralf Küsters and Tomasz Truderung. Using ProVerif to Analyze Protocols with Diffie-Hellman Exponentiation. In *IEEE Computer Security Foundations Symposium (CSF '09)*, pages 157–171. IEEE, 2009.

[Low95] Gavin Lowe. An attack on the needham-schroeder public-key authentication protocol. *Information processing letters*, 56(3):131–133, 1995.

[Low97a] Gavin Lowe. Casper: A compiler for the analysis of security protocols. In *IEEE 10th Computer Security Foundations Workshop (CSF '97). Proceedings*, pages 18–30. IEEE, 1997.

[Low97b] Gavin Lowe. A hierarchy of authentication specification. In *IEEE 10th Computer Security Foundations Workshop (CSF '97)*, pages 31–44, 1997.

[Low99] Gavin Lowe. Towards a completeness result for model checking of security protocols. In *Journal of Computer Security*, pages 96–105. Society Press, 1999.

[Mea96] Catherine Meadows. The nrl protocol analyzer: An overview. *The Journal of Logic Programming*, 26(2):113–131, 1996.

[MGV13] Sebastian Mödersheim, Thomas Groß, and Luca Viganò. Defining privacy is supposed to be easy. In *Logic for Programming, Artificial Intelligence, and Reasoning - 19th International Conference, LPAR-19, Stellenbosch, South Africa, December 14-19, 2013. Proceedings*, pages 619–635, 2013.

[Mil97] Jonathan Millen. CAPSL: Common authentication protocol specification language. Technical report, Technical Report MP 97B48, The MITRE Corporation, 1997.

[MK14] Sebastian Mödersheim and Georgios Katsoris. A sound abstraction of the parsing problem. In *IEEE The 27th Computer Security Foundations Symposium (CSF '14)*, pages 259–273. IEEE, 2014.

[MM82] Alberto Martelli and Ugo Montanari. An efficient unification algorithm. *ACM Trans. Program. Lang. Syst.*, 4(2):258–282, 1982.

[MM01] Jonathan Millen and Frederic Muller. Cryptographic protocol generation from CAPSL, December 2001.

[Möd09] Sebastian Mödersheim. Algebraic Properties in Alice and Bob Notation. In *Proceedings of the The Forth International Conference on Availability, Reliability and Security, ARES 2009, March 16-19, 2009, Fukuoka, Japan*, pages 433–440. IEEE, 2009.

[Möd10] Sebastian Mödersheim. Abstraction by set-membership: verifying security protocols and web services with databases. In *Proceedings of the 17th ACM Conference on Computer and Communications Security, CCS 2010, Chicago, Illinois, USA, October 4-8, 2010*, pages 351–360, 2010.

[Möd11] Sebastian Mödersheim. Diffie-Hellman without Difficulty. In *Formal Aspects of Security and Trust - 8th International Workshop, FAST 2011, Leuven, Belgium, September 12-14, 2011. Revised Selected Papers*, pages 214–229, 2011.

[Möd12a] Sebastian Mödersheim. Deciding Security for a Fragment of ASLan. In *Computer Security - ESORICS 2012 - 17th European Symposium on Research in Computer Security, Pisa, Italy, September 10-12, 2012. Proceedings*, pages 127–144. Springer, 2012.

[Mod12b] Paolo Modesti. *Verified Security Protocol Modeling and Implementation with AnBx*. Phd-thesis, Università Ca' Foscari Venezia, 2012.

[Mod14] Paolo Modesti. Efficient Java Code Generation of Security Protocols Specified in AnB/AnBx. In *Security and Trust Management - 10th International Workshop, STM 2014, Wroclaw, Poland, September 10-11, 2014. Proceedings*, pages 204–208, 2014.

[MS01] Jonathan K. Millen and Vitaly Shmatikov. Constraint solving for bounded-process cryptographic protocol analysis. In *CCS*, pages 166–175. ACM, 2001.

[MSCB13] Simon Meier, Benedikt Schmidt, Cas Cremers, and David Basin. The tamarin prover for the symbolic analysis of security protocols. In *Computer Aided Verification*, pages 696–701. Springer, 2013.

[MSDL99] J Mitchell, A Scedrov, N Durgin, and P Lincoln. Undecidability of bounded security protocols. In *Workshop on Formal Methods and Security Protocols*. Citeseer, 1999.

[MV09a] Sebastian Mödersheim and Luca Viganò. Secure pseudonymous channels. In *Computer Security - ESORICS 2009, 14th European Symposium on Research in Computer Security, Saint-Malo, France, September 21-23, 2009. Proceedings*, pages 337–354, 2009.

[MV09b] Sebastian Mödersheim and Luca Viganò. The Open-source Fixed-point Model Checker for Symbolic Analysis of Security Protocols. *Foundations of Security Analysis and Design V*, pages 166–194, 2009.

[MV14] Sebastian Mödersheim and Luca Viganò. Sufficient conditions for vertical composition of security protocols. In *9th ACM Symposium on Information, Computer and Communications Security, ASIA CCS '14, Kyoto, Japan - June 03 - 06, 2014*, pages 435–446, 2014.

[NPW02] Tobias Nipkow, Lawrence C Paulson, and Markus Wenzel. *Isabelle/HOL: a proof assistant for higher-order logic*, volume 2283. Springer Science & Business Media, 2002.

[NS78] Roger M. Needham and Michael D. Schroeder. Using encryption for authentication in large networks of computers. *Communications of the ACM*, 21(12):993–999, 1978.

[Qua13] Jose Nuno Quaresma. *On Building Secure Communication Systems*. Phd-thesis, Technical University of Denmark, 2013.

[RS03] Ramaswamy Ramanujam and S. P. Suresh. Tagging makes secrecy decidable with unbounded nonces as well. In *FST TCS 2003: Foundations of Software Technology and Theoretical Computer Science, 23rd Conference, Mumbai, India, December 15-17, 2003, Proceedings*, pages 363–374, 2003.

[RS11] Mark Dermot Ryan and Ben Smyth. Applied pi calculus. In *Formal Models and Techniques for Analyzing Security Protocols*. IOS Press, 2011.

[RT03] Michaël Rusinowitch and Mathieu Turuani. Protocol insecurity with a finite number of sessions and composed keys is NP-complete. *Theor. Comput. Sci.*, 299, 2003.

[Sha08] Robin Sharp. *Principles of protocol design*. Springer, 2008.

[TH05] Benjamin Tobler and Andrew CM Hutchison. Generating network security protocol implementations from formal specifications. In *Certification and Security in Inter-Organizational E-Service*. Springer, 2005.

[THG99] F. Javier Thayer, Jonathan C. Herzog, and Joshua D. Guttman. Strand spaces: Proving security protocols correct. *Journal of Computer Security*, 7(1):191–230, 1999.

[Tur06] Mathieu Turuani. The CL-Atse protocol analyser. *Term Rewriting and Applications*, pages 277–286, 2006.


الحمد لله رب العالمين